

L'Assembler per lo ZX SPECTRUM

T. Woods

L'Assembler per lo
ZX SPECTRUM

L'Assembler per lo ZX SPECTRUM

T. Woods

McGRAW-HILL Book Company GmbH

Amburgo · New York · St Louis · San Francisco · Auckland · Bogotá · Città del Guatemala · Città del Messico · Johannesburg · Lisbona · Londra · Madrid · Montreal · Nuova Delhi · Panama · Parigi · San Juan · San Paolo · Singapore · Sydney · Tokyo · Toronto

Titolo originale: *Learn and use Assembly Language on the ZX Spectrum*

Realizzazione editoriale: Edigeo snc, via Ozanam 10a, 20129 Milano
Traduzione: Elena Fossati
Grafica di copertina: Valentina Boffa
Composizione e stampa: Litovelox, Trento

ISBN 88-7700-003-1

1ª edizione Giugno 1984

Indice

Prefazione all'edizione italiana

Il testo di Woods costituisce un completo corso di Assembler che consente al lettore di familiarizzare con questo non facile linguaggio.
La presenza di numerosi listati e figure aiuta a comprendere più facilmente i concetti espressi.
I programmi presentati sono stati assemblati usando lo *ZX Spectrum Machine Code Assembler*, verificati e listati con una stampante a getto d'inchiostro per una maggiore leggibilità.
I programmi sono stati scritti per lo ZX Spectrum 16K senza elementi aggiuntivi. L'uso della Interface 1 e dei Microdrive sposta il punto d'inizio dell'area di memoria destinata ai programmi BASIC (PROG) e, poiché i programmi sono assemblati a PROG 15, ciò richiede opportune modifiche degli indirizzi delle istruzioni ORG e RANDOMIZE.
Per l'uso dello ZX Spectrum 48K occorre inoltre tenere presente la diversa posizione di RAMTOP e, conseguentemente, dell'assemblatore.

Prefazione

Questo libro si rivolge a chi non si sente soddisfatto dei risultati ottenuti sullo Spectrum con l'uso del BASIC. I programmi in linguaggio Assembler permettono il controllo del microprocessore — il "cervello" del calcolatore — così da sfruttare appieno le sue prestazioni.
Due sono le ragioni che fanno preferire la scrittura dei programmi in Assembler anziché in BASIC; in primo luogo si ottiene un vantaggio notevole per quanto riguarda la velocità di elaborazione e secondariamente il programma occupa molta meno memoria.
L'aumento di velocità è difficile da valutare finché non si prova: vi permetterà di creare grafici che si muovono in modo continuo alla velocità prescelta. Normalmente i grafici animati dovranno essere rallentati per poter essere visibili. In circostanze normali, un programma in Assembler è almeno venti volte più veloce dello stesso programma in BASIC, fino ad arrivare ad un massimo di 200 volte.
L'impiego di una quantità minore di memoria ci consente di poter memorizzare un numero maggiore di dati. L'Assembler permette inoltre un accurato controllo del modo in cui i dati sono immagazzinati in memoria. Adottando tecniche avanzate di memorizzazione, si può, in alcune circostanze, ridurre lo spazio che questa operazione esige ad una frazione di quanto richiesto dal BASIC.
Questo libro vi mostra come scrivere programmi in linguaggio Assembler. Prima che un programma in Assembler possa essere eseguito, deve essere tradotto in codice-macchina, che è il solo linguaggio compreso direttamente dal calcolatore. Poiché un programma in codice-macchina è costituito interamente da una serie di numeri è praticamente impossibile scriverlo direttamente in codice-macchina.

Il linguaggio Assembler è il linguaggio più vicino al codice-macchina e di facile comprensione.
La traduzione viene realizzata mediante l'uso di un programma assemblatore. Può essere effettuata anche manualmente ma in genere, fatta eccezione per brevi programmi, è un'operazione molto noiosa e soggetta a errori. Esistono numerosi programmi assemblatori disponibili per lo Spectrum; tutti i programmi di questo libro sono stati realizzati grazie allo ZX Spectrum Machine Code Assembler che presenta tutte le opzioni necessarie per formulare e tradurre programmi in Assembler. È un programma ideale sia per chi affronta per la prima volta l'Assembler sia per chi è già un esperto programmatore.
L'Assembler è semplicemente un altro linguaggio di programmazione e non dovrebbe essere più difficile da imparare del BASIC. Come tutti i linguaggi, l'unico modo per impararlo è di scrivere moltissimi programmi. Al termine di ogni capitolo ho proposto, come prova per il lettore, un problema di programmazione, studiato per essere alla portata delle capacità del lettore ma che richiede una certa riflessione.
Da ultimo, ringrazio mia moglie Marilyn per aver dattilografato il testo e per avermi sopportato pazientemente durante la stesura del libro. Ringrazio anche mio figlio Richard per il suo aiuto nel caricamento e nella messa a punto dei programmi.

TONY WOODS

Capitolo

All'interno dello Spectrum 1

1.1 Il computer Spectrum

Lo Spectrum è un valido home e personal computer. Può essere usato per una notevole varietà di impieghi; può divertire, può insegnare ed è in grado persino di gestire un'azienda.
La gran parte delle operazioni realizzabili da un computer è caratterizzata da tre fasi separate:

1. Inserimento delle informazioni nel computer
2. Elaborazione di queste informazioni
3. Esposizione dei risultati dell'elaborazione

La figura 1.1 mostra un tipico sistema Spectrum. La maggior parte delle informazioni viene inserita nel computer tramite una tastiera, ma ciò può anche avvenire da cassette di nastro magnetico o Microdrive su cui le informazioni erano state salvate precedentemente, o tramite joystick e sensori. Una volta introdotte nel computer, le informazioni vengono registrate nella memoria interna.
Il momento successivo, dopo l'input, è l'elaborazione di tali informazioni per fornire i risultati richiesti. Questa operazione è eseguita dall'unità centrale di elaborazione del computer, il microprocessore, costituito da un unico frammento di silicio; quello usato nello Spectrum è uno Z80A, lo stesso microprocessore presente in buona parte dei personal computer.
Infine i risultati ottenuti da questa elaborazione sono visualizzati dal computer; di solito ciò avviene sullo schermo di un televisore, ma potreb-

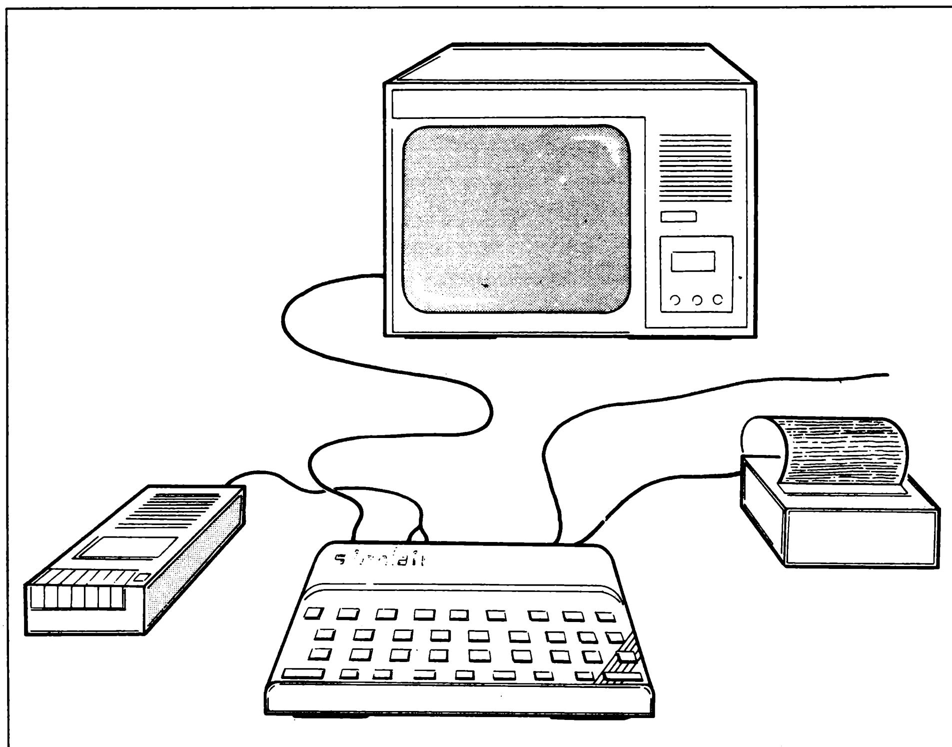


Figura 1.1

bero anche presentarsi sotto forma di suono prodotto da altoparlanti o di testi su una stampante o di segnali per il controllo diretto di apparecchiature esterne, come le valvole di un sistema centrale di riscaldamento.

1.2 I linguaggi del computer

Il computer, prima di qualsiasi utilizzo, necessita di istruzioni che regolino il suo funzionamento, normalmente definite programma.
Molti sono i linguaggi di programmazione, la maggior parte dei quali ha il fine di adattarsi ad un particolare modo di impiego del computer. La figura 1.2 mostra come esempio un piccolo programma scritto in linguaggio COBOL. Questo linguaggio è proposto per essere usato nei programmi commerciali e la sua caratteristica predominante è la facilità con cui può essere letto da persone che non siano necessariamente programmatori.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. ESEMPIO.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT MOVIMENTI ASSIGN TO DISCO.
    SELECT STAMPA ASSIGN TO STAMPA.
DATA DIVISION.
FILE SECTION.
FD  MOVIMENTI; LABEL RECORD IS STANDARD;
    DATA RECORD IS REGISTRO-MOV.
01  REGISTRO-MOV.
    02  REPARTO         PICTURE 99999.
    02  IMPORTO         PICTURE 99999999.
FD  STAMPA; LABEL RECORD IS STANDARD;
    DATA RECORD IS RIGA.
01  RIGA.
    02  TESTO            PICTURE X(18).
    02  SREPARTO         PICTURE 99999.
    02  STOTALE          PICTURE 99999999.
WORKING-STORAGE SECTION.
77  ATTUALE         PICTURE 99999.
77  TOTALE               PICTURE 99999999.
PROCEDURE DIVISION.
APRE.
    MOVE 'TOTALE PER REPARTO' TO TESTO.
    OPEN INPUT MOVIMENTI.
    OPEN OUTPUT STAMPA.
    READ MOVIMENTI.
INIZIA.
    MOVE REPARTO TO ATTUALE.
    MOVE IMPORTO TO TOTALE.
LOOP.
    READ MOVIMENTI.
    IF REPARTO NOT EQUAL TO ATTUALE GO TO LISTA.
    ADD IMPORTO TO TOTALE.
    GO TO LOOP.
LISTA.
    MOVE TOTALE TO STOTALE.
    MOVE ATTUALE TO SREPARTO.
    WRITE RIGA.
    IF REPARTO NOT EQUAL TO '99999' GO TO INIZIA.
    CLOSE MOVIMENTI.
    CLOSE STAMPA.
```

Figura 1.2

Un altro linguaggio di programmazione, certamente conosciuto, è il BASIC che è stato studiato come linguaggio ad uso generale, facile da apprendere e da usare. Entrambi, il COBOL ed il BASIC, sono stati progettati per un uso specifico ed i programmi prodotti in questi linguaggi "girano" su un'ampia gamma di calcolatori. Poiché non sono vincolati ad un computer in particolare, non utilizzano completamente le numerose possibilità presenti nell'unità centrale di elaborazione; ciò vale anche per lo Spectrum. Questo libro vi spiega come programmare lo ZX80 in un Assembler nato espressamente per il microprocessore Z80, che ne sfrutta tutte le potenzialità. Questo linguaggio proprio perché è stato progettato per un particolare tipo di microprocessore viene definito "di basso livello".

1.3 Il microprocessore Z80

Ci sono molti tipi di microprocessore, ognuno dei quali ha il proprio linguaggio Assembler. In questo libro ci occupiamo esclusivamente dell'Assembler Z80, usato dallo Spectrum. Dal momento che si vuole usufruire direttamente di tutte le prestazioni offerte dal microprocessore, sarà opportuno osservarne la struttura interna.
La figura 1.3 mostra lo schema a blocchi di un microprocessore che consiste essenzialmente in un certo numero di differenti sezioni unite da una linea di dati (bus) a 8 bit, il che significa che numeri binari di otto cifre possono essere trasferiti ed elaborati dal microprocessore.
Per il programmatore i componenti più importanti sono i registri. La figura 1.4 presenta la loro struttura nello Z80. I registri sono aree di memoria in grado di contenere un singolo elemento di informazione e vengono impiegati per conservare temporaneamente i dati, durante o nell'attesa dell'elaborazione.
Tutti i dati o le informazioni vengono memorizzati ed impiegati all'interno del computer come numeri binari; chi non avesse familiarità con essi, troverà una spiegazione nel prossimo capitolo. Il microprocessore Z80 possiede alcuni registri che trattano numeri binari di otto cifre, ed altri di sedici. Parlando di numeri binari, si usa adottare il termine bit, che è l'abbreviazione di "binary digit" (cifra binaria). I registri che conservano numeri binari di otto cifre, si chiamano registri ad 8 bit, e quelli di 16 cifre, registri a 16 bit. Lo Z80 è un microprocessore ad 8 bit, il che significa che la maggior parte dei dati usati ha una struttura ad 8 bit. Poiché è un 8 bit avanzato, presenta anche alcune possibilità per l'elaborazione di numeri a 16 bit.

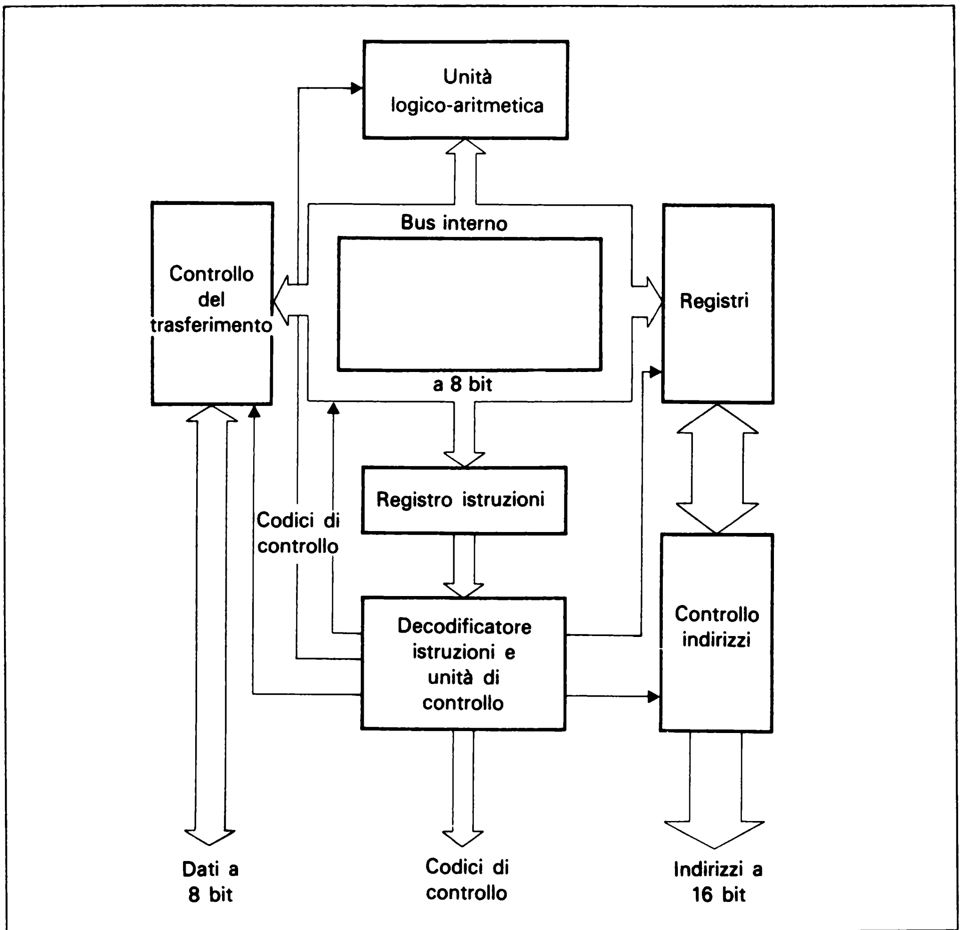


Figura 1.3

1.4 I registri

Il registro A, chiamato anche accumulatore, è quello più importante. È un registro ad 8 bit ed è impiegato soprattutto nei calcoli matematici ed in altre operazioni di elaborazione, come i confronti. Per esempio, nella somma di due numeri, uno di essi è posto nel registro A, l'altro è aggiunto a questo ed il risultato viene lasciato nel medesimo registro.
Il registro F, chiamato anche "registro dei flag", è molto particolare e serve per indicare le diverse condizioni prodotte dall'avvenuta elaborazione. Dopo aver eseguito un calcolo, ad esempio l'elementare addizione

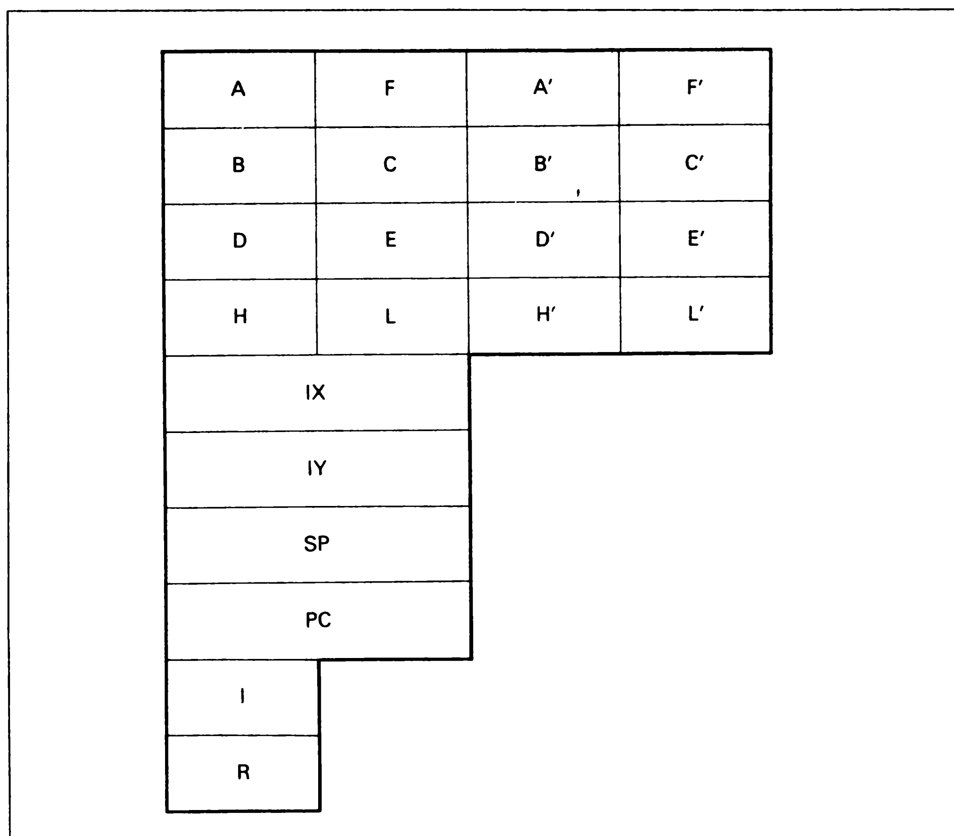


Figura 1.4

del precedente capoverso, alcuni bit del registro F indicheranno se il risultato era positivo o negativo e se era o no zero. Questi possono essere poi verificati tramite istruzioni del programma simili all'"IF" nel BASIC. Il registro F non è utilizzato direttamente dal programmatore.
I registri B, C, D, E, H ed L sono tutti registri ad 8 bit aventi degli scopi generici; possono anche essere usati in coppia come registri a 16 bit, chiamati BC, DE, HL. Sebbene siano di uso generale e possano essere praticamente intercambiabili, nell'uso comune tendono invece ad avere un impiego più specifico, che verrà chiarito più avanti; ad esempio, la coppia di registri HL è spesso impiegata come puntatore di una specifica locazione nella memoria del computer.
Il registro PC è a 16 bit ed il suo fine è quello di segnalare al computer in che punto della memoria può essere rintracciata la successiva istruzione del programma.

Il registro SP, a 16 bit, è impiegato per gestire una catasta (stack), un pratico strumento di programmazione che sarà spiegato in seguito.
I due registri IX e IY sono a 16 bit e sono chiamati registri indice; essi vengono impiegati quando il programmatore desidera far uso di tabelle o liste di dati. Nello Spectrum, il registro IY non dovrebbe essere utilizzato per nessun motivo dall'utente poiché è già impiegato dal sistema stesso.
Infine ci sono due registri ad 8 bit, I e R, che hanno applicazioni particolari nel funzionamento del computer e che sono usati raramente dai programmatori.

Capitolo

I bit e i byte 2

2.1 I numeri binari

All'interno del computer, tutti i dati sono composti da impulsi elettrici. Per comodità, si può dire che la presenza di un impulso elettrico può essere indicata dal numero 1 e la sua assenza dal numero 0, cioè che tutti i dati nel computer possono essere rappresentati da numeri composti solamente da zero ed uno, chiamati numeri binari.
Prima di considerare i numeri binari è utile riesaminare i principi della numerazione decimale che usiamo quotidianamente. Ad esempio, il numero 764 impiega tre numeri 7, 6 e 4 ma, poiché sono nell'ordine indicato, si sa che sette sta per sette centinaia, sei per sei decine e quattro per quattro unità. Potremmo così scriverlo:

$$764 = 7\times100+6\times10+4\times1$$

oppure

$$764 = 7\times(10\times10)+6\times(10)+4\times(1)$$

ciò significa che ogni posizione a sinistra è dieci volte maggiore di quella immediatamente precedente. I numeri decimali utilizzano un multiplo di dieci per ogni spazio, perciò sono definiti numeri in base 10. Qualsiasi altro numero può avere la stessa funzione; i numeri generalmente impiegati nei computer sono in base 2, cioè numeri binari, e in base 16, cioè numeri esadecimali. I numeri binari sono i più importanti in quanto nel computer tutti i dati si presentano in questa forma; in qualche caso sono usati numeri esadecimali che costituiscono un semplice sistema compat-

to di scrittura dei numeri binari. Entrambi possono essere analizzati in cifre separate come un numero decimale. Il numero binario 1011 può essere scritto:

$$1011B = 1\times8+0\times4+1\times2+1\times1$$

oppure

$$1011B = 1\times(2\times2\times2)+0\times(2\times2)+1\times(2)+1\times(1)$$

ed il numero esadecimale 945:

$$945H = 9\times256+4\times16+5\times1$$

oppure così:

$$945H = 9\times(16\times16)+4\times(16)+5\times(1)$$

La lettera B nel primo esempio ha lo scopo di mostrare che il numero è binario. Allo stesso modo, anche la H nel secondo esempio sottolinea che il numero è esadecimale (Hexadecimal; l'Assembler ZX Spectrum si rifà alla convenzione comune di porre il simbolo del dollaro all'inizio di un numero per mostrare che è esadecimale, ad esempio $1234 è uguale a 1234H).
Qualsiasi numero decimale può essere scritto impiegando le cifre da 0 a

Decimali	Binari	Esadecimali
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Figura 2.1

9, cioè da zero fino a uno meno del valore della base. In ogni base numerica occorrono simboli per i numeri compresi tra lo zero ed uno meno del valore della base.
Nella numerazione binaria si può scrivere qualsiasi numero, utilizzando solo le cifre uno e zero. I numeri esadecimali hanno bisogno di simboli compresi tra il numero zero e 15. Per i valori da 0 a 9 si usano gli stessi simboli dei numeri decimali, cioè le cifre da 0 a 9, mentre per i valori compresi tra 10 e 15 non è possibile impiegare gli stessi del sistema decimale, perché ciò potrebbe creare della confusione; si useranno perciò le lettere dalla A alla F per rappresentare tali valori. La figura 2.1 mostra le equivalenze tra numeri decimali, binari e esadecimali fino a 15.

2.2 La trasformazione delle basi numeriche

I numeri binari e quelli esadecimali possono essere trasformati in decimali trascrivendoli interamente, come mostrato all'inizio di questo capitolo, e calcolando poi i loro valori. La figura 2.2 offre degli esempi di trasformazione di un numero binario e di un numero esadecimale. La trasformazione di numeri dal sistema decimale a quello binario è invece leggermente più complicata poiché comporta una continua divisione del numero decimale per 2 e la relativa annotazione del resto. La figura 2.3 presenta questa operazione per il numero decimale 254. Si noti che il numero binario risulta dalla lettura dei resti partendo dall'ultimo trovato e procedendo in senso contrario. La trasformazione da un decimale ad un esadecimale è simile a quella dal decimale al binario, ad eccezione del divisore che sarà 16.
I numeri possono essere trasformati dal sistema binario a quello esadecimale dividendo il numero binario in gruppi di quattro bit, incominciando dalla parte destra del numero e aggiungendo degli zeri all'estremo sini-

```
01011011B = 1 × 1 + 1 × 2 + 0 × 4 + 1 × 8 + 1 × 16 + 0 × 32 + 1
            × 64 + 0 × 128
          = 1 + 2 + 0 + 8 + 16 + 0 + 64 + 0
          = 91

    5AC7H = 7 × 1 + C × 16 + A × 256 + 5 × 4096
          = 7 × 1 + 12 × 16 + 10 × 256 + 5 × 4096
          = 7 + 192 + 2560 + 20480
          = 23239
```

Figura 2.2

```
2)245(1
2)122(0
2)61(1
2)30(0
2)15(1
2)7(1
2)3(1
2)1(1
  0
```

Figura 2.3

stro se ciò si rivela necessario per completare un gruppo di quattro bit. Ogni gruppo viene poi trasformato nell'equivalente cifra esadecimale, come presentato nella figura 2.1.
Il passaggio dal sistema esadecimale a quello binario è eseguito sostituendo ogni cifra esadecimale con l'equivalente gruppo di quattro bit, come da figura 2.1. La figura 2.4 mostra alcuni esempi di tali trasformazioni.

```
0011   1011   0111   1010
  3      B      7      A
0011101101111010B = 3B7AH
  5      A      C      7
0101   1010   1100   0111
5AC7H = 0101101011000111B
```

Figura 2.4

2.3 Bit e byte

L'unità base dei dati nello Spectrum è definita byte; un byte è un numero binario ad 8 bit. Il valore di un byte può essere impiegato per rappresentare numeri, caratteri o anche istruzioni di programma. È importante rendersi conto che lo stesso numero binario può avere significati differenti, a seconda della sua posizione nella memoria. La rappresentazione dei caratteri sarà trattata nel capitolo 5.

La rappresentazione numerica sta ad indicare che gli elementi del byte sono considerati come numeri binari; perciò un byte contenente le cifre binarie 01101101 rappresenta il numero 1101101B, che nel sistema decimale vale 109. La serie di numeri che può essere inclusa in un singolo byte è compresa fra 00000000B e 11111111B, cioè da 0 a 255 in decimali; può rappresentare numeri positivi ed è nota come rappresentazione di numeri senza segno per distinguerla dal successivo metodo che comprende entrambi i numeri, sia positivi che negativi. Infatti si ha spesso bisogno di utilizzare numeri con valori negativi e per questo scopo si impiega un differente metodo di rappresentazione, definito "sistema di complemento a due" o numeri con segno. La base di questo metodo è che il numero di cifre binarie con cui qualsiasi numero è rappresentato nel calcolatore è sempre lo stesso. Il numero di cifre è stabilito dal programmatore a seconda delle dimensioni del problema da risolvere; esso deve comunque essere un multiplo di otto e spesso è proprio otto, cioè un byte. Con questo metodo si dà il normale valore ad ogni posizione di bit nel

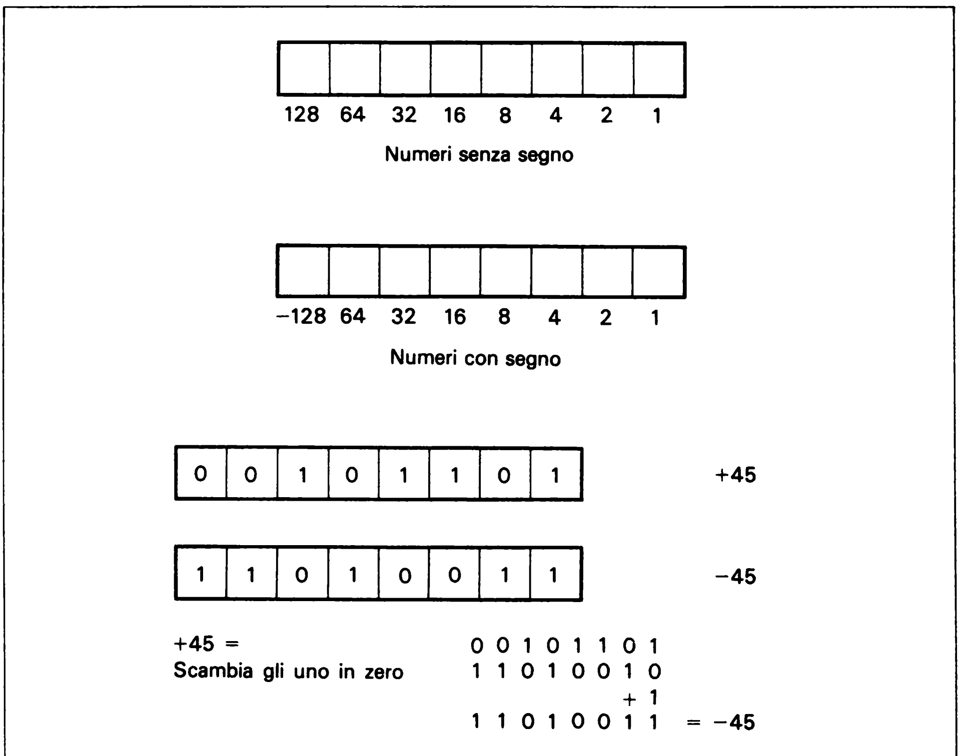


Figura 2.5

numero, fatta eccezione per l'ultimo bit a sinistra che assume il negativo del normale valore. La figura 2.5 indica il valore di ogni bit in un numero ad 8 bit, sia in versione con segno che senza. In quella con segno la posizione della cifra a sinistra rappresenta il numero −128 e tutti gli altri numeri assumono il loro normale valore. Usando questi valori, un numero positivo occuperà le posizioni di sette cifre positive, e la cifra all'estrema sinistra sarà sempre zero; i valori negativi saranno ottenuti aggiungendo quanto basta a −128. La figura 2.5 illustra degli esempi di numeri con segno positivo e negativo. Il computer non utilizza tale metodo per trovare il complementare. Per ricavare il numero negativo dal positivo, rappresenta il numero positivo in base binaria e trasforma tutti gli zero in uno e tutti gli uno in zero, aggiungendo infine uno a questo numero. Anche questo è riscontrabile nella figura 2.5.

2.4 La memoria

Si può immaginare la memoria dello Spectrum come suddivisa in un certo numero di scomparti, chiamati locazioni, ognuno dei quali contiene un numero binario ad 8 bit. Dal momento che ogni locazione comprende 8 bit, si possono considerare nel loro insieme come byte di memoria anche se, per precisione, questo termine dovrebbe essere usato solo per riferirsi ai contenuti di un'area di memoria.
I byte della memoria sono numerati in successione, partendo da zero. Il numero assegnato ad ogni byte rappresenta il suo indirizzo, così come le case di una strada sono numerate per poter essere individuate; nello stesso modo, esso permette di ritrovare ogni particolare byte nella memoria. Nell'Assembler è il programmatore che stabilisce quali zone nella memoria saranno impiegate per la memorizzazione di dati. Sovente il programmatore deve poter accedere ai dati memorizzati in una precisa zona di memoria. In Assembler un modo conciso per scrivere "il valore del byte di memoria, il cui indirizzo è" è quello di racchiudere l'indirizzo in parentesi in modo tale che, ad esempio, (23637) si riferisce non al numero 23637 ma al valore del byte di memoria alla locazione 23637.
All'interno dello Spectrum ci sono due tipi differenti di memoria, chiamati ROM e RAM. ROM, che significa "memoria a sola lettura" (Read Only Memory), è la memoria che contiene un programma permanentemente inserito nel computer; non può essere modificata dal programmatore, ma il programma, o parte di esso, può essere utilizzato in altri programmi. La ROM nello Spectrum usa le zone di memoria dall'indirizzo 0 a quello 16383. La memoria restante è RAM, che significa "memoria ad accesso casuale" (Random Access Memory). Questa può essere modificata dal programmatore, sebbene ci siano delle zone che è preferibile lasciare im-

mutate. Nessun danno effettivo può essere arrecato dalla modifica di alcuni byte di memoria di RAM anche se, in alcuni casi, ciò potrebbe bloccare il computer. Il fatto in sè non è così grave come potrebbe sembrare, poiché per recuperare il controllo è sufficiente l'interruzione dell'alimentazione.
Il modello standard 16K possiede delle locazioni di memoria da 16 volte 1024 RAM (la lettera K nel gergo dei computer sostituisce il numero 1024). La RAM occupa le locazioni dall'indirizzo 16384 al 32767. Lo Spectrum 48K possiede 48K di locazioni di memoria RAM dall'indirizzo 16384 al 65535. Questa memoria è la più grande che possa essere usata sullo Spectrum senza dover usare tecniche speciali per indirizzare le locazioni.

Capitolo

La programmazione in linguaggio Assembler

3

3.1 Il linguaggio Assembler

Prima che qualsiasi programma entri in funzione nel computer, tutte le sue istruzioni devono essere tradotte in una serie di numeri binari.
L'unità centrale di elaborazione del computer, nel nostro caso il microprocessore Z80, accetta solo istruzioni di programma scritte in numeri binari. La funzione principale della ROM, nello Spectrum, consiste nella trasformazione di istruzioni BASIC in numeri binari, noti come codice-macchina. Poiché il BASIC non è stato scritto in modo specifico per il microprocessore Z80, la traduzione in codice-macchina è relativamente inefficiente e molto lenta per un computer.
Scrivere un programma e introdurlo nello Spectrum direttamente in sistema binario è impossibile, è invece possibile convertire i numeri binari in decimali, e, in seguito, mediante l'istruzione POKE inserire il tutto nella memoria. Vale la pena osservare che uno degli effetti dell'istruzione POKE è quello di trasformare il numero da decimale in binario prima della sua introduzione nella memoria.
La figura 3.1 rappresenta un breve programma in decimale; leggendo questo programma non viene fornita nessuna idea circa la funzione di una qualsiasi delle istruzioni. Introdurre i programmi in questo modo è piuttosto noioso e facilmente soggetto ad errori, considerando che qualsiasi sbaglio diviene molto difficile da scoprire. Non bisogna inoltre scordare che questo è solo un breve programma e si può facilmente immaginare quali problemi nascerebbero se fosse più lungo.
Al fine di sfruttare appieno le potenzialità del microprocessore Z80 è ne-

cessario impiegare un linguaggio di programmazione simile al codice-macchina usato dal computer ma facile da leggere e comprendere; il linguaggio che soddisfa queste richieste è l'Assembler.
Il principale scopo dell'Assembler consiste proprio nel rendere maggiormente comprensibile il codice-macchina del computer. Per esempio la prima istruzione nella figura 3.1, il decimale 120, viene convertita nel numero binario 01111000B. Ciò rappresenta l'istruzione necessaria per caricare i contenuti del registro B nell'accumulatore. In Assembler tale istruzione sarebbe rappresentata con:

LD A,B

Date uno sguardo all'appendice A, che offre una lista completa di tutte le istruzioni accettate dal microprocessore Z80; non bisogna preoccuparsi se non si riesce a comprendere parte di esse poiché verranno spiegate in un secondo momento. La figura 3.2 presenta il programma della figura 3.1 scritto in Assembler: si noterà che, benché il numero 120 sia la prima istruzione nel programma in codice-macchina, il suo equivalente in Assembler LD A,B non è la prima istruzione nel programma ma è preceduta dalle istruzioni aventi lo scopo di dare informazioni al programma che lo traduce dall'Assembler in codice-macchina.
L'Assembler fa uso di piccoli gruppi di lettere per indicare l'operazione da svolgere; tali lettere vengono scelte in modo tale da aiutare il programmatore a ricordare quale operazione sta compiendo; ad esempio per caricare i dati in un registro, le lettere usate sono LD (da Load) ed esse diventano ADD (da Addition) in caso di somma.

Indirizzo	Contenuto
23760	50
23761	0
23762	120
23763	33
23764	208
23765	92
23766	70
23767	128
23768	50
23769	209
23770	92
23771	201

Figura 3.1

```
10 REM go
12 REM org 23760
13 REM !riserva spazio per i numeri
16 REM Num2;defb 50
18 REM Result;defb 0
19 REM !carica il primo numero
20 REM ld a,b
21 REM !carica il secondo
22 REM ld hl,Num2
24 REM ld b,(hl)
25 REM !esegue la somma
26 REM add a,b
27 REM !conserva il risultato
28 REM ld (Result),a
29 REM !ritorna al BASIC
30 REM ret
32 REM finish
```

Figura 3.2

Durante la scrittura di istruzioni in Assembler, la posizione degli spazi, virgole o parentesi presenti nelle istruzioni, è molto importante. Ad esempio un Assembler come quello dello ZX Spectrum accetterebbe un'istruzione come LD (32500),A ma non LD(32500),A oppure LD (32500)A. Se, traducendo un programma, si ottiene un messaggio di errore, il primo indispensabile passo da compiere è quello di controllare attentamente la struttura dell'istruzione.

3.2 Un programma tipo

È giunto il momento di dare uno sguardo ad un programma completo in Assembler e di studiare l'applicazione di questo sullo Spectrum. Il programma rappresentato nella figura 3.3 è un semplice programma di rinumerazione. Esso rinumera solo le istruzioni e non eventuali richiami contenuti nei GOTO e GOSUB. Una volta giunti ad un livello più avanzato, forse vorrete tentare di scrivere una routine più completa. La maggior parte dei programmi in Assembler impiegati sullo Spectrum verranno eseguiti usando un apposito programma BASIC; tale programma, per la routine di rinumerazione è indicato nella figura 3.4.
A questo punto è possibile che non riusciate a comprendere la maggior

```
 10 REM go
 20 REM org 23760 32500
 22 REM !programma per rinumerare un programma in BASIC
 23 REM !
 25 REM !trova l'inizio del programma
 30 REM ld hl,(23635)
 35 REM ld de,0;push de;!azzera DE
 40 REM ld bc,10;push bc;!assegna l'incremento del numero di linea
 50 REM Loop;pop bc;pop de
 55 REM ex de,hl
 58 REM !calcola il numero di linea
 60 REM add hl,bc
 70 REM ex de,hl
 75 REM !conserva il numero di linea
 80 REM ld (hl),d
 90 REM inc hl
100 REM ld (hl),e
110 REM push de;push bc;inc hl
115 REM !trova la lunghezza della linea
120 REM ld c,(hl)
130 REM inc hl
140 REM ld b,(hl)
150 REM inc hl
160 REM add hl,bc
165 REM !trova il numero della linea successiva
170 REM ld d,(hl)
180 REM inc hl
190 REM ld e,(hl)
200 REM dec hl
205 REM !cerca la linea 9000 per verificare la fine del programma
210 REM ld a,d
220 REM sub 35
230 REM jp m,Loop
240 REM ld a,e
250 REM sub 40
260 REM jp m,Loop
270 REM ret
280 REM finish
```

Figura 3.3

```
9000 CLEAR 32500
9010 LOAD "figura 3.3"CODE
9020 RANDOMIZE USR 32500
9030 LIST
```

Figura 3.4

parte delle istruzioni del programma in Assembler. Può essere più interessante dare un'occhiata al programma BASIC affiancato che è molto breve. La prima linea assegna la variabile di sistema RAMTOP per mantenere una parte di memoria libera per il programma in codice-macchina. La linea seguente è usata per inserire il programma in codice-macchina nella memoria così riservata.
La linea che segue è la più importante del programma; essa contiene l'istruzione che fa eseguire il programma in codice-macchina. L'effetto di tale istruzione è quello di eseguirlo come se fosse una subroutine del programma BASIC. Non bisogna preoccuparsi se non si riesce a comprendere come funzionano le subroutine poiché il tutto verrà ampiamente spiegato in seguito. L'istruzione finale lista il programma rinumerato al fine di evidenziare il risultato.

3.3 Le istruzioni

Generalmente tutte le istruzioni in codice-macchina usate dal microprocessore sono formate da due parti. La prima, chiamata l'*operazione*, istruisce il computer sull'azione da svolgere, mentre la seconda, chiamata l'*operando*, gli indica quali dati usare. La prima istruzione nella figura 3.5 è il numero decimale 120 trasformato nel numero binario 01111000. Le prime cinque cifre di questo numero, cioè 01111, rappresentano il codice dell'operazione "caricare nell'accumulatore" e le tre cifre finali, 000, indicano l'operando, in questo caso il valore del registro B. L'istruzione completa è "caricare nell'accumulatore il valore del registro B".
Un ulteriore sguardo all'appendice A mostrerà che le istruzioni non sono tutte di uguale lunghezza: alcune usano solo un byte, altre due, altre ancora tre o quattro. Queste differenze sono principalmente dovute ai diversi modi di precisare gli operandi, conosciuti come metodi di indirizzamento e ampiamente descritti in seguito.
La figura 3.5 presenta le diverse istruzioni di differenti lunghezze sia in Assembler che in codice-macchina.

Assembler	Codice-macchina	
LD A,B	01111000B	120
LDIR	11101101B	237
	10110000B	176
JP 32000	11000011B	195
	00000000B	0
	01111101B	125
LD (32000),IX	11011101B	221
	00100010B	34
	00000000B	0
	01111101B	125

Figura 3.5

3.4 Dall'Assembler al linguaggio macchina

Un programma scritto in Assembler deve essere tradotto in codice-macchina prima che venga svolto dal computer. Diversamente dai programmi in BASIC, che sono tradotti ed eseguiti una linea per volta, tutto il programma in Assembler viene tradotto prima dell'esecuzione. Questo comporta che una zona di memoria oltre alla memoria usata per contenere il programma in Assembler, deve essere impiegata per memorizzare il programma in codice-macchina. La figura 3.6 illustra un uso tipico della memoria con un programma in Assembler.
Il metodo più vantaggioso per realizzare la traduzione è quello di impiegare un programma assemblatore; esso effettuerà la traduzione in codice, verificherà eventuali errori del programma e lo introdurrà in memoria. Tutti i programmi presenti in questo libro sono stati preparati usando lo ZX Spectrum Machine Code Assembler; questo programma estende realmente il raggio d'azione del vostro Spectrum. L'appendice B descrive l'uso di questo assemblatore. Se non è disponibile un assemblatore, i programmi possono essere tradotti manualmente; come farlo è descritto dettagliatamente nell'appendice D.
Nel momento in cui il programma è stato trasformato in codice-macchina, può essere eseguito dal computer e si potranno incontrare e correggere ulteriori errori; è molto più difficile individuarli che nel BASIC perché i programmi in codice-macchina non producono alcun messaggio di errore. Un altro importante problema dei programmi in linguaggio macchina è che il tasto BREAK non produce alcun effetto, a me-

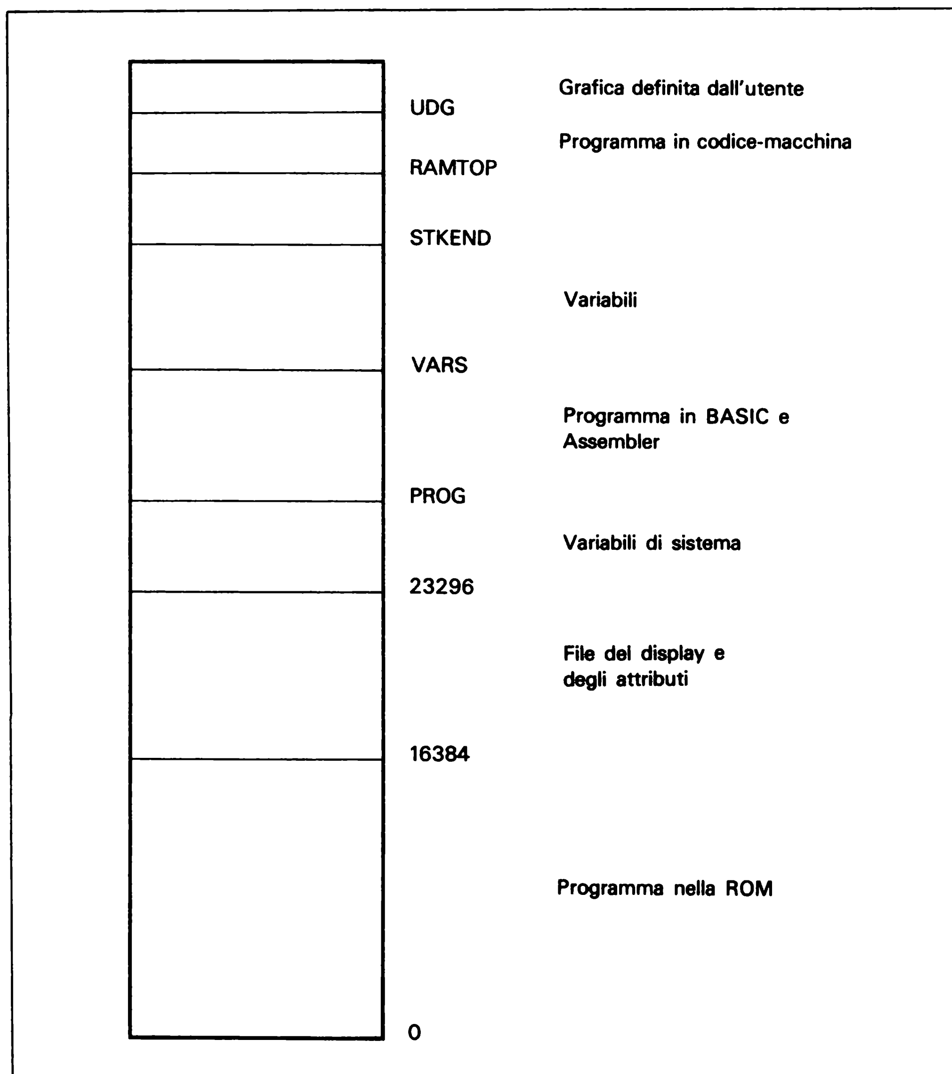


Figura 3.6

no che sia scritto in modo specifico nel programma; un metodo per farlo verrà spiegato più avanti. Questo provoca il fatto che, se un programma in codice-macchina incontra un loop infinito — il che è solitamente indicato dall'apparente inoperosità del computer — il solo modo di riacquistarne il controllo è quello di spegnere e di riaccendere il computer. Si scoprirà così che è molto utile abituarsi a registrare i programmi prima di farli girare. Così facendo se si è cosí sfortunati da cadere in un loop

infinito non si dovrà ribattere tutto il programma. Dato che lo Spectrum è studiato principalmente per i programmi in BASIC, tutti i programmi in linguaggio macchina vengono eseguiti come subroutine del sistema BASIC. Il normale metodo di avviamento di un programma in binario consiste nell'istruzione o comando RANDOMIZE USR XXXX, dove XXXX indica la locazione di partenza nella memoria del programma in codice-macchina. La funzione USR è un tipo speciale di chiamata di una subroutine. Essa è impiegata per le subroutine in linguaggio macchina allo stesso modo in cui il GOSUB è usato per le subroutine in BASIC. Poiché tutti i programmi in binario si svolgono come subroutine del sistema BASIC, devono terminare tutti con l'istruzione RET (che significa "rientro da una subroutine") cosicché il controllo viene restituito al BASIC.

3.5 La memorizzazione dei programmi

Il programmatore in Assembler lavora molto più vicino al microprocessore, del programmatore che usa il BASIC. Questo gli offre un migliore controllo ed impiego delle caratteristiche del microprocessore; tuttavia significa anche che al programmatore non è consentito dare niente per scontato e deve eseguire una certa quantità di compiti svolti automaticamente nel BASIC. Uno di questi compiti è quello di decidere dove, nella memoria del computer, deve essere immagazzinato il programma in codice, così da poterlo facilmente elaborare. Principalmente tre sono le differenti aree di memoria utilizzabili, tutte con i loro vantaggi e svantaggi. La zona più sicura per memorizzare i programmi si trova in testa alla memoria. Il programma può essere qui inserito e la variabile di sistema RAMTOP assegnata alla locazione di memoria esattamente al di sotto dell'inizio di programma; questo viene così protetto da sovrascrittura dal BASIC. Il principale svantaggio nell'uso di questa area di memoria riguarda il fatto che il programma in codice ed il programma di supporto in BASIC devono essere registrati e caricati separatamente. Se avete esperienza di codice-macchina su un computer ZX81, conoscerete il metodo di memorizzare i programmi mediante l'istruzione REM al principio di un programma BASIC. Ciò può essere applicato anche allo Spectrum e presenta il vantaggio che il programma in codice viene automaticamente registrato con il programma BASIC. Lo svantaggio di questo metodo è che l'inizio di un programma BASIC non ha una posizione fissa sullo Spectrum, ma dipende dalle periferiche collegate. L'inizio dell'area di programma è ricavato dalla variabile di sistema PROG. Infine, il programma può essere memorizzato nell'area libera di memoria fra la catasta operativa e quella di macchina; questa area è definita dalla variabile di sistema STKEND.

Probabilmente il miglior modo di operare è quello di usare un'istruzione REM per memorizzare il programma mentre ci state lavorando, ma quando il programma è in funzione dovrebbe essere memorizzato in testa alla memoria e salvaguardato dai programmi in BASIC, riassegnando la variabile RAMTOP mediante l'istruzione CLEAR. La figura 3.7 illustra queste tre aree di memoria.

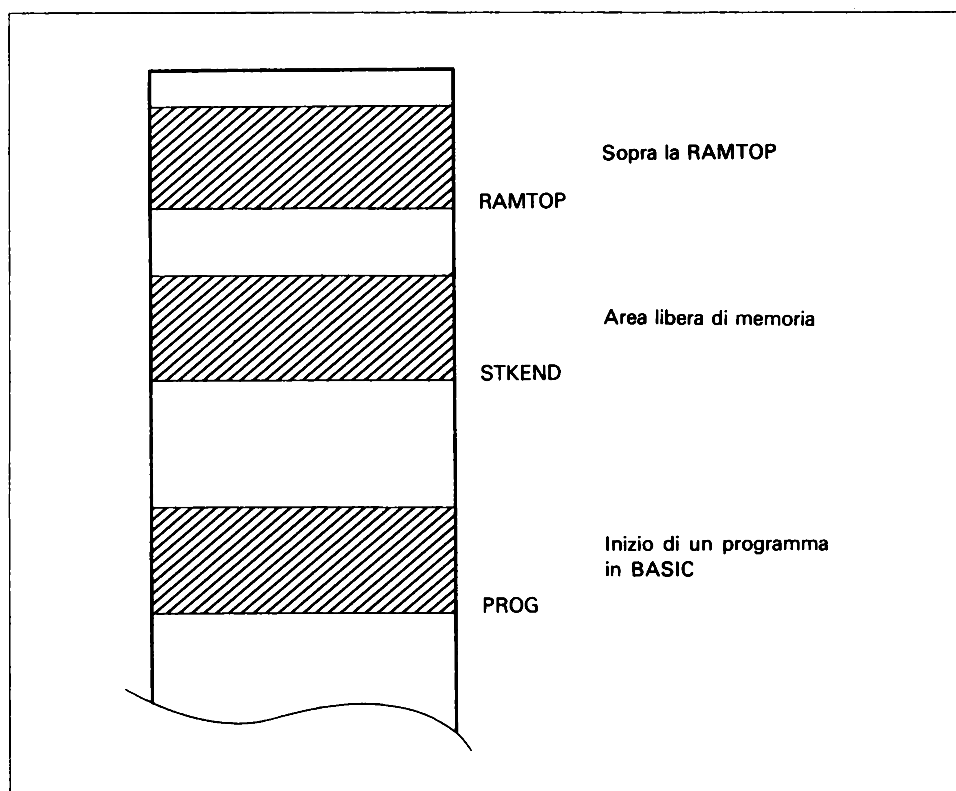


Figura 3.7

3.6 Le subroutine

Le subroutine sono una tecnica di programmazione molto importante, specialmente per il programmatore in Assembler. In considerazione della loro importanza, questo paragrafo descriverà il motivo per cui sono usate ed il loro modo d'impiego. Non si soffermerà su come operano, poiché questo sarà il compito di uno dei prossimi capitoli.

```
 10 REM go
 20 REM org 23766
 30 REM !uso delle subroutine
 32 REM !carica il primo numero
 35 REM ld a,(23760)
 38 REM !chiama la subroutine
 40 REM call Multen
 43 REM !conserva il risultato
 45 REM ld (23763),a
 48 REM !ripete il ciclo per il  secondo numero
 50 REM ld a,(23761)
 55 REM call Multen
 60 REM ld (23764),a
 62 REM !ripete il ciclo per il  terzo numero
 65 REM ld a,(23762)
 70 REM call Multen
 75 REM ld (23765),a
 80 REM ret;!fine del programma principale
 85 REM !inizio della subroutine
 90 REM Multen;add a,a;!raddoppia
 95 REM ld b,a;!conserva il dato parziale
100 REM add a,a;!raddoppia di nuovo
105 REM add a,a;!raddoppia ancora una volta (a=8*a)
110 REM add a,b;!8*a+2*a=10*a
115 REM ret;!ritorna al programma principale
120  REM finish
```

Figura 3.8

```
  1 REM 0000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000
  10 INPUT a,b,c
  20 LET x=23760
  30 POKE (x+0),a
  40 POKE (x+1),b
  50 POKE (x+2),c
  60 RANDOMIZE USR 23766
  70 FOR i=3 TO 5
  80 PRINT PEEK (x+i)
  90 NEXT i
```

Figura 3.9

Perché si usano le subroutine? Si supponga di scrivere un programma formato da due o più gruppi di istruzioni che svolgono la medesima azione. È una perdita di tempo ripetere in continuazione la stessa serie di istruzioni all'interno del programma ed è più opportuno scriverle una sola volta, come se fossero un blocco a sé stante. Questa è una subroutine. Solitamente le subroutine vengono scritte separatamente dopo il programma principale. Quando questo è in esecuzione, ogni qualvolta è necessaria una subroutine, viene utilizzata una speciale istruzione di salto per darle il via. Al termine di questa, un'altra istruzione di salto riporta indietro il controllo all'istruzione del programma principale successiva a quella da cui ha avuto origine il salto alla subroutine. L'Assembler per lo Spectrum impiega l'istruzione CALL per far effettuare il salto alla subroutine. CALL deve essere seguita dalla locazione di memoria del primo byte della subroutine. Si verificherà più avanti che è possibile assegnare nomi, o label, ai byte di memoria; la label può essere utilizzata invece dell'indirizzo numerico. La figura 3.8 è un esempio di breve programma che utilizza una subroutine per moltiplicare per 10 una lista di numeri. La figura 3.9 è un breve programma in BASIC che impiega il programma della figura 3.8. A questo livello i dettagli del programma sono irrilevanti; è possibile osservare che la subroutine viene attivata dall'istruzione:

```
CALL MULTEN
```

dove MULTEN è una label che indica il punto in cui comincia la subroutine. Quando il programma è trasformato in codice macchina, la label verrà convertita nel numero della locazione di memoria che contiene la prima istruzione della subroutine.

L'istruzione finale in qualsiasi subroutine è RET. Ciò costringe il computer a ritornare alla giusta posizione del programma principale. Tutti i programmi in binario nello Spectrum sono eseguiti mediante la funzione USR; poiché questa è un salto alla subroutine, tutti i programmi in linguaggio macchina devono terminare mediante l'istruzione RET.

Un programma può impiegare un qualsiasi numero di subroutine se l'inizio di ciascuna di esse è indicato in modo chiaro e ogni subroutine termina con un'istruzione RET. Un programma avente più di una subroutine solitamente è più facile da capire se tutte queste si trovano al termine del programma principale, una dopo l'altra.

Uno dei vantaggi delle subroutine è che possono essere usate anche se sono state scritte da altre persone, senza che sia necessario conoscerne i dettagli. Un'importante fonte di subroutine per lo Spectrum sono le routine della ROM. L'appendice G ne elenca alcune ed il loro modo d'impiego; non è una lista completa delle routine utili ma solamente di alcune delle più facili da usare.

Capitolo

Alcune semplici istruzioni 4

4.1 I dati nel computer

Nel corso di questo capitolo daremo uno sguardo alle istruzioni in Assembler che consentono lo spostamento dei dati all'interno del computer e l'esecuzione di operazioni elementari sui numeri memorizzati in un byte. Tutti i dati usati in un programma in Assembler devono essere spostati dal programmatore nelle loro corrette locazioni di memoria. Un programma in Assembler è costituito in gran parte da istruzioni che muovono i dati dall'una all'altra locazione di memoria.
Le istruzioni di aritmetica elementare impiegano un'altra grande percentuale del programma; maggiormente usate sono quelle che incrementano o decrementano di uno il valore in una locazione di memoria (un registro o un byte di memoria).

4.2 I registri di caricamento

I dati, in attesa o durante l'elaborazione, vengono conservati nei registri dell'unità centrale di elaborazione. Si può accedere alle informazioni contenute in un registro in circa metà del tempo necessario per trasferirle dalla memoria principale. Naturalmente solo un limitato numero di dati può essere contenuto nei registri.
Un valore numerico può essere inserito direttamente in uno qualsiasi dei singoli registri a 8 bit utilizzando l'istruzione:

LD r,n

dove n indica il valore da introdurre nel registro, mentre r uno qualsiasi dei registri a 8 bit A, B, C, D, E, H o L. Per esempio l'istruzione:

LD D,20

metterà l'equivalente binario del numero 20 nel registro D; i contenuti di tale registro appariranno come nella figura 4.1. Normalmente il valore di un registro viene considerato come un numero intero e positivo e dovrà essere compreso nella serie contenuta in 8 bit; deve perciò variare tra 0 e 255. Il computer può anche essere programmato per trattare il valore in un registro come un numero con segno o complementare binario. In questo caso può contenere i valori da −128 a +127. I numeri con segno sono stati descritti nel capitolo 2.

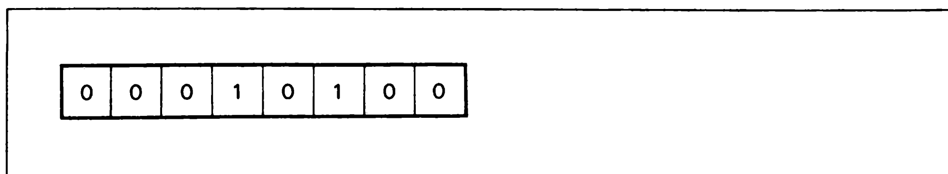


Figura 4.1

Il registro A, chiamato anche accumulatore, è il più importante per il programmatore. Tutte le operazioni logiche e la maggior parte di quelle aritmetiche implicano l'uso dell'accumulatore e generalmente il risultato dell'elaborazione rimane in esso.
Poiché i registri vengono spesso impiegati come locazioni di memoria per i dati in attesa di elaborazione, subentra l'esigenza di trasferire i dati fra i registri. L'istruzione che consente tale spostamento è:

LD rl,r2

L'effetto di questa istruzione consiste nell'immissione di una copia del valore situato nel registro r2 nel registro rl. Il valore del registro r2 non viene modificato dall'istruzione. I registri rl e r2 possono appartenere ad uno qualsiasi di quelli a 8 bit A, B, C, D, E, H o L. Ad esempio, se il registro A contiene il valore 47 e il C il valore 127, l'istruzione LD C,A mette il valore 47 nel registro C e lascia il valore 47 in A. La figura 4.2 illustra i contenuti di entrambi i registri prima e dopo l'istruzione.

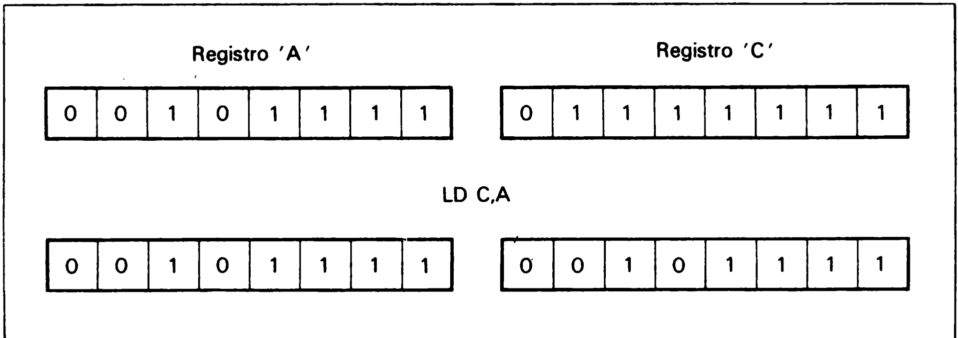


Figura 4.2

4.3 Incrementi e decrementi

Le operazioni aritmetiche più utilizzate sono l'incremento e il decremento di uno del valore di un registro o di una locazione di memoria. Ci sono due istruzioni per effettuare queste operazioni che si presentano in questo modo:

INC r e DEC r

Tali istruzioni hanno l'effetto di aumentare o diminuire di uno il valore in uno dei registri a 8 bit A, B, C, D, E, H o L, o in una delle coppie di registri a 16 bit BC, DE, HL. Queste istruzioni possono anche essere impiegate per aumentare o diminuire il valore di una locazione di memoria, inserendo prima l'indirizzo della stessa nella coppia di registri HL ed impiegando poi questa come indice. Ad esempio, se la coppia di registri HL contiene il valore 32000 e la locazione di memoria 32000 contiene il valore 58, l'istruzione INC(HL) aumenterà il valore nella locazione di memoria 32000 di 1, raggiungendo così 59. Bisogna ricordare che le parentesi sono un metodo conciso per riferirsi al valore contenuto in una locazione di memoria. La figura 4.3 illustra l'uso della coppia di registri HL come indice.

4.4 Trasferimenti in memoria

La maggior parte dei dati usati dal programmatore verranno conservati nella memoria e dovranno essere trasferiti ai e dai registri nel processore centrale prima e dopo l'elaborazione. L'accumulatore, o registro A, è il

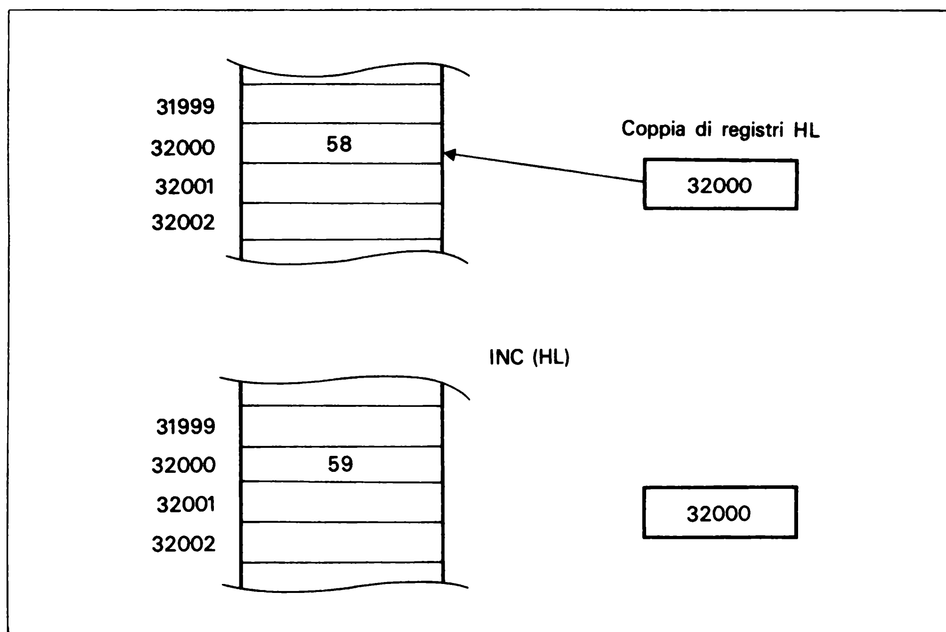


Figura 4.3

registro più adatto ad essere usato per l'elaborazione ed è il più flessibile in caso di trasferimento di dati da e nella memoria.
Tutte le istruzioni che impiegano i dati nella memoria contengono l'indirizzo della locazione di memoria oppure utilizzano una coppia di registri come indice. L'istruzione più semplice per il trasferimento dei dati al registro A è:

LD A,(nn)

dove nn indica l'indirizzo della locazione di memoria contenente le informazioni. Non bisogna dimenticare che, nonostante una cella di memoria possa contenere un solo byte (8 bit di dati), l'indirizzo di una locazione di memoria richiede invece un numero binario a 16 bit. Perciò 'nn' è un numero che va da 0 a 65535. Allo stesso modo, i dati possono essere spostati dal registro A alla memoria mediante l'istruzione:

LD (nn),A

Non sempre è possibile specificare direttamente in ogni istruzione l'indirizzo della locazione di memoria. Sovente l'indirizzo viene calcolato da

una parte precedente del programma ed è contenuto in una delle coppie di registri a 16 bit. Le istruzioni,

LD A,(rr) e LD(rr),A

dove rr è una delle coppie di registri BC, DE, HL, utilizzano il valore nel registro a 16 bit come indirizzo della locazione di memoria coinvolta nello spostamento. Come abbiamo precedentemente affermato, l'effetto delle istruzioni di trasferimento consiste nel trascrivere i dati dalla memoria all'accumulatore o viceversa, senza modificare il valore del dato originario.
Sappiamo che la maggior parte dei registri secondari nell'unità centrale di elaborazione viene usata per scopi specifici e perciò il primo che prenderemo in considerazione è la coppia di registri HL; questa costituisce un registro a 16 bit il cui uso principale è di indicare le locazioni nella memoria. In altre parole è il registro generalmente usato per contenere l'indirizzo dei dati oggetto di spostamento da e nella memoria. Usando le istruzioni:

LD r,(HL) E LD (HL),r

i dati possono essere spostati fra la memoria ed uno qualsiasi dei registri a 8 bit. Impiegando ancora la coppia di registri HL come puntatore, può venire direttamente caricata una locazione di memoria con un valore numerico. Il modello dell'istruzione è:

LD (HL),n

dove n deve essere un numero compreso fra 0 e 255.

4.5 L'addizione e la sottrazione

Diamo ora uno sguardo ad alcune semplici regole di aritmetica. Il microprocessore Z80 contiene delle istruzioni che ci permettono di aggiungere o di sottrarre i valori nell'accumulatore. Le istruzioni per effettuare tali operazioni, usando direttamente un valore numerico, sono:

ADD A,n e SUB n

dove n è un numero compreso tra 0 e 255.
È importante sottolineare che la forma di queste due istruzioni è diversa; questo perché, sebbene il registro A sia il solo a 8 bit in grado di essere

usato con l'istruzione ADD, l'addizione a 16 bit può essere eseguita mediante l'uso della coppia di registri HL, ma la sottrazione, utilizzando l'istruzione SUB, può essere svolta solo tramite i valori a 8 bit e l'accumulatore. Ciò significa che nell'istruzione ADD deve essere indicato l'uso del registro A, mentre ciò non è necessario con l'istruzione SUB perché viene usato in ogni caso il registro A. Bisogna precisare che l'istruzione SUB sarebbe errata se fosse scritta allo stesso modo dell'istruzione ADD; non è facoltativo evitare il riferimento al registro A.
In esso rimane il risultato del calcolo, quando vengono svolte delle addizioni o sottrazioni mediante l'impiego dell'accumulatore.
Ad esempio l'effetto delle istruzioni:

```
LD A,52
SUB 19
ADD A,22
INC A
```

è quello di inserire il valore 52 nell'accumulatore, sottrargli 19 lasciando il valore 33, aggiungere 22 a quest'ultimo giungendo al valore 55 ed infine aumentarlo di uno per lasciare il valore 56 nell'accumulatore.

```
 10 REM go
 20 REM org 23760
 25 REM !acquisisce la locazione del primo dato
 30 REM ld hl,32500
 35 REM !carica il primo numero
 40 REM ld a,(hl)
 50 REM inc hl
 55 REM !carica il secondo
 60 REM ld b,(hl)
 65 REM !li somma
 70 REM add a,b
 80 REM ld b,a
 85 REM !raddoppia il risultato
 90 REM add a,a
 95 REM !conserva il risultato
100 REM ld (hl),b
110 REM dec hl
120 REM ld (hl),a
130 REM finish
```

Figura 4.4

Il valore di uno dei registri a 8 bit, compreso l'accumulatore, può essere aggiunto o sottratto al valore nell'accumulatore mediante le istruzioni:

ADD A,r e SUB r

Impiegando HL come puntatore, anche un valore nella memoria può essere aggiunto o sottratto dal valore nell'accumulatore.
La figura 4.4 illustra un breve segmento di programma comprendente alcune istruzioni oggetto di questo capitolo. La prima istruzione mette il valore 32500 nella coppia di registri HL. Il valore nella locazione di memoria 32500 viene poi trascritto nel registro A mediante l'uso di HL come puntatore. L'istruzione seguente aumenta di uno il valore in HL fino a giungere a 32501 e poi il valore di questa locazione di memoria viene trascritto nel registro B. A questo punto i due valori vengono sommati e il risultato viene trasferito dall'accumulatore nel registro B. In seguito il valore nell'accumulatore si raddoppia ed infine i valori nei registri A e B vengono ricopiati in memoria.
Se all'inizio del programma la locazione di memoria contiene il valore 20 e la 32501 contiene 15, quali saranno i valori in queste due locazioni di memoria alla fine del programma? Se avete qualche dubbio a questo proposito, la figura 4.5 indica i valori nei registri e nelle locazioni di memoria dopo ciascuna delle istruzioni del programma.

Istruzione	HL	A	B	Locazione 32500	Locazione 32501
LD HL,32500	32500	?	?	20	15
LD A,(HL)	32500	20	?	20	15
INC HL	32501	20	?	20	15
LD B,(HL)	32501	20	15	20	15
ADD A,B	32501	35	15	20	15
LD B,A	32501	35	35	20	15
ADD, A,A	32501	70	35	20	15
LD (HL),B	32501	70	35	20	35
DEC HL	32500	70	35	20	35
LD (HL),A	32500	70	35	70	35

Figura 4.5

4.6 La scrittura di un programma

Il programma illustrato nella figura 4.6 è una subroutine utile per chi scrive programmi di giochi; è un tipo di programma che farà scorrere a sinistra una linea di caratteri sullo schermo.
Questo è un programma interessante da esaminare poiché in gran parte è costituito da spostamenti di dati e aritmetica elementare.
La prima istruzione è una direttiva, vale a dire un'istruzione al programma assemblatore di riservare una locazione della memoria. Inizia il programma vero e proprio quando si carica nella coppia di registri HL l'indirizzo della locazione di memoria del primo byte di dati per la linea che

```
 10 REM go
 20 REM org 23760
 22 REM Temp;defb 0
 25 REM !programma per lo scroll laterale della prima riga
 30 REM ld hl,16384;!inizio della riga
 40 REM ld c,8;!numero delle linee per riga
 50 REM Loopa
 55 REM ld a,(hl)
 60 REM ld (Temp),a;!conserva il primo carattere
 70 REM ld b,31;!numero dei caratteri da spostare
 75 REM !loop per traslare una linea di un carattere
 80 REM Loopb
 85 REM inc hl
 90 REM ld a,(hl)
100 REM dec hl
110 REM ld (hl),a
120 REM inc hl
130 REM djnz Loopb
145 REM !pone il primo carattere alla fine della riga
150 REM ld a,(Temp)
160 REM ld (hl),a
165 REM !trova la linea successiva della riga
170 REM ld de,225
180 REM add hl,de
190 REM dec c
200 REM jp nz,Loopa
210 REM ret
220 REM finish
```

Figura 4.6

deve scorrere; nell'esempio indicato è la prima dall'alto. Utilizzando i dati forniti in uno dei capitoli seguenti a proposito del file di display, si potrà modificare questo programma per far scorrere qualsiasi linea. Per meglio comprendere il programma bisogna sapere che ogni linea di caratteri sullo schermo è memorizzata come otto blocchi di 32 byte e c'è un blocco di 224 byte fra ciascuno di essi.
Osservando il programma si dovrebbero capire tutte le istruzioni tranne la DJNZ Loopb e JP NZ, Loopa. JP NZ significa "saltare se la precedente istruzione aritmetica ha dato un risultato diverso da zero" e DJNZ è una combinazione delle istruzioni: DEC B e JP NZ.

4.7 Le label

Quando si programma in BASIC si fa spesso uso di istruzioni che si riferiscono ad altre nel programma (per esempio quando si usano le istruzioni GOTO). Nel BASIC ogni linea del programma possiede un numero che viene impiegato per il riferimento di altre istruzioni a tale linea. I programmi in linguaggio macchina non hanno nessun numero di linea quando sono conservati nella memoria. Le istruzioni nel programma che si riferiscono ad altre istruzioni nello stesso programma, lo fanno fornendo l'indirizzo della locazione di memoria che contiene il primo byte dell'istruzione.
Calcolare la locazione di memoria che accoglie il primo byte di un'istruzione particolare è molto difficile e costringe ad un'inutile perdita di tempo. Nell'Assembler questo problema è superato grazie alle label, vale a dire ai nomi dati a particolari istruzioni presenti nel programma. Quando ci si riferisce ad un'istruzione tramite un'altra, viene usata la sua label. Nel momento in cui un programma viene assemblato in codice-macchina, il programma trasforma automaticamente le label nell'esatto indirizzo di memoria.
Diversamente dal BASIC sono assegnate delle label solo alle istruzioni alle quali altre si riferiscono. Il programmatore sceglie le label all'interno delle regole imposte dal programma assemblatore.

4.8 Esercizio

A questo livello i programmi possibili sono molto limitati e devono essere delle semplici subroutine richiamabili da un programma BASIC.
Si scriva una subroutine in Assembler che moltiplichi due numeri sommando il primo a se stesso per una quantità di volte indicata dal secondo

numero. L'operazione 4×3 equivale a 4+4+4. Per scrivere questo programma sarà necessaria l'istruzione: JP NZ, label.
Si assembli il programma nella parte alta della memoria e si definisca la RAMTOP per lasciare tre locazioni di memoria protette prima di iniziare il programma. Ad esempio nello Spectrum 16K si ponga RAMTOP a 32499 e si assembli il programma per cominciare dalla locazione 32503 mediante l'uso di "org 32503". I due numeri da moltiplicare insieme saranno nelle locazioni di memoria 32500 e 32501 e il risultato verrà collocato nella locazione di memoria 32502.
Si scriva un programma in BASIC che acquisisca due numeri e li memorizzi nelle locazioni 32500 e 32501; quindi si chiami la subroutine in linguaggio macchina ed infine si visualizzi il risultato ricavandolo dalla locazione 32502. Molto probabilmente avrete avuto modo di notare in un programma precedente che la coppia di registri HL può essere caricata direttamente con un numero. (Cfr. Fig. 4.4).

Capitolo

Il salto 5

5.1 Perché il salto?

Una delle cose che sappiamo della programmazione in BASIC è che ben pochi programmi partono dalla prima istruzione e arrivano all'ultima eseguendole una sola volta e seguendo l'ordine dei loro numeri di linea. Infatti tutti i programmi, tranne i più semplici, contengono dei salti che cambiano l'ordine in cui vengono eseguite le istruzioni. Ci sono due tipi di istruzioni di salto: quelle incondizionate e quelle condizionate.

5.2 I salti incondizionati

Nel BASIC l'istruzione di salto incondizionato è: "GOTO al numero di linea". In Assembler l'istruzione è:

```
JP nn
```

dove nn indica l'indirizzo di una locazione di memoria. L'istruzione ha l'effetto di considerare l'indirizzo nn come il primo byte della istruzione successiva. Generalmente in Assembler il numero nn viene sostituito da una label; ad esempio, in un programma assemblato per partire dalla locazione 32500 nella memoria e data la label "Start" alla prima istruzione, le istruzioni:

JP 32500 e JP Start

sono equivalenti.
I salti incondizionati non sono molto utili di per sé perché generano sovente dei loop senza uscita, come indicato nella figura 5.1. Sfortunatamente, nello Spectrum, se il programma entra in un loop senza uscita si può fermarlo solo spegnendo il computer e ricominciando da capo.

```
10 REM go
20 REM org 23760
30 REM ld b,1;ld a,32
40 REM Loop;add a,b
50 REM push af;rst 16;pop af
60 REM jr Loop
70 REM finish
```

Figura 5.1

L'istruzione JP consente al computer di saltare in qualunque punto all'interno della memoria accessibile della macchina; nella maggior parte dei programmi, JP salta a istruzioni a pochissimi byte di distanza dall'istruzione di salto. Il modo per trarre vantaggio da questo è un secondo salto incondizionato che permetta di realizzare questi brevi salti. Questa è l'istruzione di salto relativo che occupa solo due byte di memoria invece dei tre richiesti dalla istruzione JP.
Il modello dell'istruzione è:

JR n

dove n è un numero nell'intervallo da −128 e +127. Nuovamente ci si può riferire all'istruzione da eseguire dopo il salto tramite una label, cioè JR label. L'assemblatore calcolerà il numero di byte da saltare e sostituirà la label con il relativo salto. Quando si usano le label bisogna essere certi che il salto richiesto, espresso in binario, sia contenibile nella locazione di memoria, altrimenti si potrebbe incorrere in errore al momento dell'assemblaggio del programma. In caso di dubbio è consigliabile impiegare l'istruzione JP.

5.3 I flag

Fra i registri nel microprocessore Z80 ce n'è uno a 8 bit di cui non abbiamo ancora parlato: è il registro F. Esso non è impiegato per la raccolta di dati ma è un insieme di bit separati, la gran parte dei quali è usata per conservare informazioni riguardanti il risultato delle istruzioni logiche o aritmetiche. La figura 5.2 presenta i nomi dati a ciascun bit. Poiché sono usati per indicare il risultato di un'istruzione precedente, sono chiamati "flag". Due bit importanti del registro F sono il "flag di segno" e il "flag di zero".

Se un'operazione logica o aritmetica dà luogo ad un risultato negativo, il flag di segno è posto ad uno, mentre un risultato positivo o uguale a zero

S	Z	X	H	X	P/V	N	C

Flag		Confronto				
S	Segno	P	Positivo	o	M	Zero o negativo
Z	Zero	Z	Zero	o	NZ	Non zero
H	Riporto intermedio	—				
P/V	Parità/overflow	PO	Dispari	o	PE	Pari
N	Addizione/sottrazione	—				
C	Riporto	C	Riporto	o	NC	Non riporto
X	Bit non utilizzato					

Figura 5.2

	Flag	
Programma	Segno	Zero
LD A,10	?	?
SUB 11	1	0
LD B,A	1	0
INC B	0	1
INC B	0	0

Figura 5.3

pone il flag a zero. Se un'istruzione genera zero, il flag zero è posto ad uno; tutti i risultati diversi da zero, sia positivi che negativi, pongono il flag a zero. Osserveremo meglio il resto dei flag nel corso di questo libro. La figura 5.3 illustra un frammento di programma con il valore del flag di segno e del flag di zero dopo l'esecuzione di ciascuna istruzione.

5.4 I salti condizionati

Dalla programmazione in BASIC si constaterà che i salti incondizionati hanno di per sé un impiego piuttosto limitato; essi sono quasi sempre utilizzati con le istruzioni che rendono un salto dipendente dai risultati di un confronto; questo tipo di salto è chiamato condizionato.
Quando un programma incontra un'istruzione di salto condizionato, esso continua con l'istruzione successiva oppure salta ad un'altra nel programma. Il salto viene eseguito soltanto se la condizione è verificata; in Assembler la condizione è indicata da uno dei bit presenti nel registro dei flag. La figura 5.4 illustra un breve programma che impiega un'istruzione di salto condizionato per uscire da un loop. Il programma somma

```
 10 REM go
 20 REM org 23760
 30 REM equ 32550 Inizio
 40 REM ld hl,Inizio
 45 REM ld b,0
 50 REM !inizio del loop
 60 REM Loop;ld a,(hl)
 70 REM !controlla se e' uguale a zero
 75 REM sub 0
 80 REM jr z,Fine
 90 REM add a,b
100 REM ld b,a
105 REM inc hl
110 REM jr Loop
120 REM Fine;ld a,b
127 REM !conserva il risultato
130 REM ld (hl),a
135 REM ret
140 REM finish
```

Figura 5.4

dei numeri nelle locazioni consecutive di memoria fino a quando incontra una locazione contenente il valore zero; questa è poi impiegata per accogliere la somma. Il modello dell'istruzione di salto condizionato è:

JP c,addr

dove la variabile è la condizione da esaminare mentre addr è la locazione di memoria che racchiude l'istruzione da eseguire se si verifica la condizione. Naturalmente l'indirizzo verrà sostituito da una label in un programma in Assembler.
Alcune condizioni che possono essere esaminate sono:

Zero	JP Z,LABEL1
Non Zero	JP NZ,LABEL2
Negativo	JP M,LABEL3
Positivo	JP P,LABEL4

Queste condizioni vengono verificate dal computer controllando il valore del bit relativo nel registro dei flag. Nel BASIC i salti possono essere effettuati in qualsiasi condizione aritmetica, ma in Assembler devono essere riscritti come confronti, usando il flag di segno e il flag di zero; tutte le condizioni aritmetiche possono venir scritte in tal modo. La figura 5.5 illustra un breve frammento di un programma in BASIC ed il suo equivalente in Assembler. Mediante una sottrazione e un confronto del flag di segno o di zero, potrà essere programmata una qualsiasi delle principali condizioni impiegate nel BASIC. La figura 5.6 evidenzia l'equivalente in Assembler delle quattro condizioni principali. Mediante una combinazione di verifiche, qualsiasi test che viene programmato in BASIC può ugualmente essere realizzato in Assembler.
È possibile anche attuare i salti condizionati relativi; le condizioni verificabili tramite un salto relativo sono limitate, perciò esso è in grado di va-

BASIC	Assembler	
10 LET A=10		LD A,10
20 PRINT A	LOOP;	RST 16
30 LET A=A+10		ADD A,10
40 IF A>100 THEN 60		SUB 100
50 GO TO 20		JP P,END
60 STOP		JP LOOP
	END;	RET

Figura 5.5

BASIC	Assembler
IF A=B THEN	SUB B JP Z,NEXT
IF A>B THEN	SUB B JP P,NEXT
IF A<B THEN	SUB B JP M,NEXT
IF A<>B THEN	SUB B JP NZ,NEXT

Figura 5.6

lutare il flag di zero ma non il flag di segno. Le istruzioni che impiegano il flag di zero sono:

JR Z,disp   e   JR NZ,disp

dove disp è il numero di byte da saltare se la condizione è verificata; esso può essere sostituito da una label mentre l'indirizzo sarà calcolato dall'Assembler.

5.5 I confronti

Il valore originario nell'accumulatore, in caso di impiego della sottrazione per azzerare i flag di condizione prima di un salto, viene modificato da tale operazione. Sovente è possibile ristabilire il valore mediante un'addizione ma può essere difficile e scomodo. Nello Spectrum il microprocessore Z80 possiede un'istruzione che supera questo problema, chiamata "di confronto" il cui modello è:

CP n

dove n è un valore a 8 bit, o il contenuto in uno dei registri a 8 bit oppure un valore nella memoria il cui indirizzo è racchiuso nella coppia di registri HL. L'istruzione consente il confronto tra il valore nell'accumulatore ed un altro a 8 bit, senza modificare il primo. L'istruzione sottrae il valore specificato dal contenuto dell'accumulatore e assegna zero o uno ai bit del registro dei flag, a seconda del risultato. Il risultato della sot-

trazione viene in seguito scartato mentre il valore originario resta nell'accumulatore.
L'impiego principale dell'istruzione di confronto è quello di determinare se il valore nell'accumulatore è quello richiesto; essa viene spesso usata dopo un input dalla tastiera per controllare se è stato inserito un carattere particolare. Dopo l'input a tastiera, l'accumulatore contiene i codici del carattere; ad esempio le istruzioni:

```
CP 65
JR Z,ALABEL
CP 13
JP NZ,LOOP
```

dopo un input a tastiera, controllano dapprima se il carattere è una A, ed in tal caso saltano all'istruzione etichettata ALABEL. Nel caso invece che non sia una A, controllano che sia stato premuto ENTER e, in caso contrario, saltano all'istruzione chiamata LOOP.
Non tutte le istruzioni modificano il registro dei flag, ad esempio l'istruzione LD A,(HL).
Sovente è utile assegnare il valore dei bit del registro dei flag secondo il valore dell'accumulatore, anche quando l'istruzione non modifica il registro dei flag. L'istruzione CP 0 può essere impiegata per determinare il valore dei flag confrontando l'accumulatore con lo zero. L'appendice A riassume le istruzioni che interessano il registro flag e i bit coinvolti. Nessuna delle istruzioni per il movimento dei dati, finora incontrate, riguarda il registro flag; se si desidera verificare il segno di un valore trascritto dalla memoria, è possibile impiegare l'istruzione CP 0, come indicato nel seguente programma:

```
LD HL,32000
NEXT; INC HL
LD A,(HL)
CP 0
JP M,NEXT
```

Questo programma continuerà a realizzare loop fino a trovare un valore positivo nella memoria.

5.6 Le pseudo-operazioni

Le pseudo-operazioni sono istruzioni in programmi in linguaggio Assembler che non creano nessun'istruzione equivalente in linguaggio macchina. I due impieghi principali delle pseudo-operazioni sono il passaggio di

informazioni all'assemblatore sul modo di tradurre il programma, sullo spazio da riservare in memoria per i dati.
Alcune pseudo-operazioni possono essere impiegate con un solo assemblatore, mentre altre vengono comprese dalla maggior parte degli assemblatori. I programmi in questo libro sono stati scritti usando lo ZX Spectrum Machine Code Assembler che impiega le istruzioni scritte nelle REM di un programma BASIC; la prima istruzione, in qualsiasi programma, deve essere la pseudo-operazione:

GO

mentre l'ultima deve sempre essere:

FINISH

Queste istruzioni sono riconosciute solo da questo assemblatore; un'altra pseudo-operazione che deve essere usata con lo ZX Spectrum Machine Code Assembler e con la maggior parte degli altri assemblatori è:

ORG nn

Questa istruzione indica all'assemblatore che la seguente deve essere caricata nella locazione di memoria nn; la maggior parte degli assemblatori la richiedono al principio del programma per indicare all'assemblatore dove posizionare il programma nella memoria; a metà di un programma modifica la locazione dell'istruzione seguente.
Lo ZX Spectrum Machine Code Assembler consente un'altra formulazione molto utile per il comando ORG, cioè:

ORG a b

dove a è la locazione di partenza nella memoria alla quale l'assemblatore carica il programma, ma esso viene tradotto come se il suo indirizzo di inizio fosse la locazione b. Nello Spectrum, la collocazione migliore per un programma in codice-macchina è in testa alla memoria, ma quando si realizzano e caricano programmi in Assembler, tale posizione è occupata dall'assemblatore. Questo modello dell'istruzione consente la traduzione del codice nella parte bassa della memoria ed il successivo trasferimento a quella alta per il suo funzionamento.
La più utile delle pseudo-operazioni è l'istruzione DEFB. Essa ordina all'assemblatore di riservare le locazioni di memoria seguenti per raccogliere valori ad 8 bit, ed è seguita dai valori da collocare in queste locazioni. Quando più di un valore segue questa istruzione è necessario lasciare uno spazio fra ognuno di essi. Solitamente l'istruzione è preceduta

```
10 REM go
15 REM org 23760
20 REM Lower;defb 97
25 REM Upper;defb 0
30 REM !
35 REM ld a,(Lower)
40 REM sub 32
45 REM ld (Upper),a
50 REM ret
55 REM finish
```

Figura 5.7

da una label e ci si può riferire alla prima locazione col nome della label. La figura 5.7 è un breve programma che illustra l'uso dell'istruzione per assegnare due locazioni di memoria. Lo scopo del programma è quello di sottrarre 32 dal numero nella locazione LOWER e di collocare poi il risultato in quella UPPER. Il programma è infatti in grado di trasformare delle lettere minuscole (LOWER) in maiuscole (UPPER).

5.7 L'output sul video

Lo Spectrum, come gran parte dei microcomputer, impiega una tecnica chiamata "memory mapping" per visualizzare le informazioni su uno schermo televisivo o su un monitor. Questa tecnica consiste nell'assegnare un'area della memoria del computer per contenere tutto quanto deve essere visualizzato. Parte dell'apposito programma è usata per attivare alcuni dei circuiti del computer al fine di trascrivere le informazioni dalla memoria all'uscita TV e da qui allo schermo.
Un modo di realizzare un'immagine sul video è quello di caricarne gli elementi di informazione in memoria ed inviarli automaticamente allo schermo. In effetti lo Spectrum riserva due aree di memoria per le immagini: una è impiegata per la raccolta delle forme da riprodurre sul video, e la seconda contiene le informazioni sul colore. Lo Spectrum è in grado di produrre grafici ad alta risoluzione e per questo la forma di ciascun carattere viene definita dal contenuto di otto locazioni di memoria separate; ciò rende molto più difficile inserire caratteri sul video caricando i loro particolari nella memoria.
Tuttavia, dato che la visualizzazione di caratteri è una necessità ricorrente, esiste una subroutine nella ROM del Sinclair che svolge tale funzione;

questa routine può essere richiamata con l'istruzione:

RST 16

A livello elementare, il suo effetto è quello di impiegare il valore nell'accumulatore come un codice di carattere e di visualizzare il carattere sul

```
 10 REM go
 15 REM org 23760
 17 REM !ciclo che si ripete 10 volte
 20 REM ld b,10
 25 REM Loopa;ld c,5
 27 REM !5 spazi all'inizio della riga
 30 REM Loopb;ld a,32
 35 REM rst 16
 40 REM dec c
 45 REM jr nz,Loopb
 47 REM !carica i codici dei caratteri
 50 REM ld a,77;!  M
 55 REM rst 16
 60 REM ld a,101;! e
 65 REM rst 16
 70 REM ld a,115;! s
 75 REM rst 16
 80 REM ld a,115;! s
 85 REM rst 16
 90 REM ld a,97;!  a
 95 REM rst 16
100 REM ld a,103;! g
105 REM rst 16
110 REM ld a,103;! g
115 REM rst 16
120 REM ld a,105;! i
125 REM rst 16
130 REM ld a,111;! o
135 REM rst 16
140 REM !va alla riga successiva
145 REM ld a,13;! ENTER
150 REM rst 16
155 REM dec b
160 REM jr nz,Loopa
165 REM ret
170 REM finish
```

Figura 5.8

```
 10 REM go
 20 REM org 23760
 30 REM !Display animato
 40 REM !Inizializzazione
 50 REM ld d,10;!numero riga
 60 REM ld e,15;!numero colonna
 90 REM ld b,127;!carattere mobile
100 REM call Print
110 REM !in alto e a destra
120 REM Ur;ld h,d
130 REM ld l,e;!conserva la posizione precedente
140 REM Ur1;dec d;!riga successiva in alto
142 REM ld a,d
145 REM !lato superiore dello schermo
147 REM cp 1
150 REM jr z,Dr1
160 REM Ur2;inc e;!colonna successiva a destra
170 REM ld a,e
180 REM cp 31
185 REM !lato dello schermo
190 REM jp z,Ul2
195 REM !cancella la vecchia e visualizza la nuova posizione
200 REM call Printer
210 REM jr Ur
220 REM !in alto e a sinistra
230 REM Ul;ld h,d
240 REM ld l,e;!conserva la posizione precedente
250 REM Ul1;dec d;!riga successiva in alto
252 REM ld a,d
255 REM !lato superiore dello schermo
257 REM cp 1
260 REM jr z,Dl1
270 REM Ul2;dec e;!colonna successiva a sinistra
272 REM ld a,e
275 REM !lato dello schermo
277 REM cp 1
280 REM jr z,Ur2
290 REM call Printer
300 REM jr Ul
310 REM !in basso e a destra
320 REM Dr;ld h,d;!conserva la posizione precedente
330 REM ld l,e
340 REM Dr1;inc d;!riga successiva in basso
350 REM ld a,d
360 REM cp 21
```

(continua)

```
365 REM !lato inferiore dello schermo
370 REM jr z,Ur1
380 REM Dr2;inc e;!colonna successiva a destra
390 REM ld a,e
400 REM cp 31
405 REM !lato dello schermo
410 REM jr z,Dl2
420 REM call Printer
430 REM jr Dr
440 REM !in basso e a sinistra
450 REM Dl;ld h,d
460 REM ld l,e;!conserva la posizione precedente
470 REM Dl1;inc d;!riga successiva in basso
480 REM ld a,d
490 REM cp 21
495 REM !lato inferiore dello schermo
500 REM jr z,Ul1
510 REM Dl2;dec e;!colonna successiva a sinistra
512 REM ld a,e
515 REM !lato dello schermo
517 REM cp 1
520 REM jr z,Dr2
530 REM call Printer
540 REM jr Dl
550 REM Print;ld a,2
552 REM push bc;push de;push hl
554 REM call 5633;!apre il canale
556 REM pop hl;pop de;pop bc
557 REM ld a,22;rst 16;!visualizza alla posizione
570 REM ld a,d;rst 16
590 REM ld a,e;rst 16
610 REM ld a,b;rst 16
630 REM ret
640 REM Printer;ex de,hl
650 REM ld b,32
660 REM call Print
670 REM ex de,hl
680 REM ld b,127
690 REM call Print
700 REM ret
710 REM finish
```

Figura 5.9

video. La routine RST 16 è molto potente perché, oltre a visualizzare un carattere a partire dal codice, è in grado anche di riconoscere e di eseguire i caratteri di controllo della stampa.
Per controllare l'input e l'output, il sistema Spectrum fa uso di una tecnica chiamata "channelling", che impiega differenti parti della ROM per ciascuno dei suoi differenti tipi di input ed output. Il sistema standard dello Spectrum utilizza quattro canali:

Canale 1 permette l'input dalla tastiera e l'output alla parte inferiore del video.
Canale 2 permette l'output della parte superiore del video ma nessun input.
Canale 3 permette l'impiego dell'area di lavoro del BASIC; operazione questa, in genere, di scarsa utilità.
Canale 4 permette l'output alla stampante ma nessun input.

Quando si accende, il computer si predispone sul canale 1 in attesa di un input dalla tastiera. Per usare gli altri canali, il numero di canale deve essere caricato nell'accumulatore e viene impiegata una subroutine della ROM per attivare quel canale. La figura 5.9 illustra l'uso di questa routine che fornisce l'output alla parte superiore del video. L'appendice E elenca i codici ASCII per i caratteri che vengono inseriti dalla tastiera e quindi stampati, mentre l'appendice F mostra i caratteri di controllo riconosciuti dalla routine RST 16. Le figure 5.8 e 5.9 illustrano dei programmi che impiegano la RST 16 per stampare nelle metà superiori ed inferiori del video. La figura 5.8 è un programma un po' più semplice di quello indicato nella figura 5.9, che mostra il modo di impiego della routine RST 16 per creare un'animazione. A questo livello, l'istruzione fornirà una sufficiente flessibilità per gran parte delle funzioni. In seguito sarà trattata la realizzazione di immagini ad alta risoluzione.

5.8 Esercizio

Si scriva un programma che visualizzi un quadrato di 6×6 caratteri "*" nel centro dello schermo. La figura 5.10 illustra l'output richiesto. Usate i caratteri di controllo per visualizzare nella posizione esatta sul video i caratteri precedenti; si consiglia l'uso dei loop per abbreviare il programma.
Come esempio più complicato, si potrebbe tentare un programma che produca l'immagine rappresentata in figura 5.11.

```
******
******
******
******
******
******
```

Figura 5.10

```
   *
  ***
 *****
*******
 *****
  ***
   *
```

Figura 5.11

Capitolo

L'uso della tastiera 6

6.1 L'input da tastiera

Molti dei nostri programmi richiedono dati da inserire nel computer tramite la tastiera. È un programma lungo e complesso quello usato per rendere il computer in grado di comprendere quando e quale tasto è stato premuto sulla tastiera ma fortunatamente è possibile usare le subroutine della ROM per svolgere tutta la difficile operazione. La subroutine principale parte dalla locazione di memoria 703 e si può usare mediante l'istruzione:

```
CALL 703
```

L'effetto della subroutine è quello di esplorare la tastiera fino a che non viene premuto un tasto e di caricare il codice del carattere nella variabile di sistema LAST-K, che si trova nella locazione di memoria 23560. Essa contiene il codice del carattere dell'ultimo tasto premuto fino a che non venga schiacciato un altro tasto o venga caricato un valore differente nella locazione. La figura 6.1 illustra una breve routine che acquisisce un carattere dalla tastiera quando viene premuto un tasto, carica il codice del carattere nell'accumulatore e lo visualizza sul video. Questa è la normale operazione del BASIC. È possibile inserire un carattere dalla tastiera senza che esso appaia sul video, annullando l'istruzione RST 16 dalla routine.

Il metodo sopradescritto è il più semplice per realizzare il processo, spesso richiesto, di immissione e di visualizzazione di un carattere sullo

```
  10 REM go
  20 REM org 23760
1000 REM Cinout;call 703
1010 REM call 4264
1020 REM cp 255;!tasto premuto?
1030 REM jr z,Cinout
1035 REM push af;!conserva durante la visualizzazione
1040 REM rst 16;!visualizza il carattere
1050 REM !attende che il tasto venga rilasciato
1060 REM Loop;call 703
1070 REM call 4264
1080 REM cp 255
1090 REM jr nz,Loop
1095 REM pop af
1100 REM ret
1110 REM finish
```

Figura 6.1

schermo. La ROM Sinclair comprende altre routine utilizzabili per l'input dalla tastiera. Se siete interessati, fate esperimenti con le routine della ROM.

6.2 I codici di carattere

È importante sottolineare che la routine di input, oggetto dell'ultimo paragrafo, carica nell'accumulatore il codice ASCII del carattere corrispondente al tasto premuto. L'appendice E elenca i normali codici di carattere che possono essere inseriti dalla tastiera mediante la routine di input. Bisogna rendersi conto che, quando si introducono dei numeri, il codice di carattere per tale numero non è lo stesso del suo valore numerico. Se vengono inseriti dei numeri per effettuare dei calcoli, bisogna assicurarsi di usare il codice di carattere ed il valore numerico in maniera appropriata. Tutti i processi di input ed output fanno uso del codice di carattere e l'aritmetica utilizza sempre il valore del carattere. Quando si usano i codici ASCII, come accade nello Spectrum, i codici di carattere delle cifre da 0 a 9 possono essere tramutati nel loro valore numerico sottraendo 48 dal codice di carattere; bisogna ricordare naturalmente di aggiungere di nuovo 48 al momento dell'uscita dei risultati.

6.3 L'input numerico

Abbiamo sinora considerato l'input e l'output dei numeri ad una cifra; in questo paragrafo saranno prese in considerazione le routine per realizzare l'input e l'output di numeri con più cifre. Tale operazione non è così diretta come può sembrare, anche se ci si limita ai numeri interi.
Se il numero da inserire è a tre cifre, ad esempio 164, bisognerà premere i tasti 1, 6, 4, chiamando per tre volte le subroutine di input e di output e caricando i codici di carattere così come sono inseriti in successive locazioni di memoria. Al termine dell'operazione di input solitamente si introduce un altro carattere, ad esempio "virgola", oppure "ENTER"; i codici di caratteri separati devono venire trasformati in valore nel registro o locazione di memoria. Il primo passo da effettuare in questa conversione è la conversione del codice di carattere nel valore numerico per ciascuna cifra. Occorre ora effettuare la moltiplicazione della prima cifra per 100, della seconda per 10 e quindi bisogna sommare i due prodotti alla terza cifra. Per il numero 164, il calcolo è:

$$1 \times 100 + 6 \times 10 + 4$$

oppure un metodo più efficiente è:

$$(1 \times 10 + 6) \times 10 + 4$$

A conversione avvenuta, la prima cifra viene moltiplicata per 10, la seconda sommata al prodotto, il risultato moltiplicato ancora per 10 ed infine la terza cifra sommata al secondo prodotto. Il grande vantaggio di questo metodo consiste nella sua capacità di essere applicato all'input di un numero avente una quantità qualsiasi di cifre.
In un precedente capitolo si è già parlato di aritmetica elementare e il lettore rammenterà che si potevano svolgere solo operazioni di addizione e di sottrazione. Non esistono istruzioni dirette in Assembler per la moltiplicazione e la divisione; un modo alquanto semplice per fare una moltiplicazione è infatti l'addizione ripetuta: ad esempio 5×3 corrisponde a $5+5+5$, allo stesso modo, 6×10 è uguale a $6+6+6+6+6+6+6+6+6+6$; può apparire piuttosto primitivo ma, usando un loop, è facilmente programmabile e notevolmente veloce.
La figura 6.2 è una subroutine che immette un numero e lo trasforma in un valore immagazzinato nella memoria. Benché la routine possa acquisire un numero di qualunque dimensione, il valore finale è contenuto in una locazione di memoria; ciò significa che il valore massimo inseribile è 255. La routine accetta qualsiasi input come una cifra fino a quando viene premuto il tasto ENTER, ma potrebbe anche essere facilmente estesa per controllare se il tasto premuto è una cifra o ENTER.

```
  10 REM go
  20 REM org 23760
  30 REM !riserva una locazione in memoria
  40 REM Numero;defb 0
  50 REM !inizia l'input del numero
  60 REM Numin;call Cinout
  70 REM !controlla se e' un ENTER
  80 REM cp 13
  90 REM ret z
 100 REM !trasforma il codice in valore
 110 REM sub 48
 120 REM ld b,a
 130 REM !richiama il totale precedente
 140 REM ld a,(Numero)
 150 REM !moltiplica per dieci
 160 REM call Multen
 170 REM add a,b
 180 REM !conserva il nuovo numero
 190 REM ld (Numero),a
 200 REM jr Numin
 210 REM !
 220 REM !subroutine per moltiplicare per dieci
 225 REM !
 230 REM Multen;add a,a
 240 REM ld c,a
 250 REM add a,a;add a,a
 270 REM add a,c
 280 REM ret
 285 REM !
 290 REM !subroutine di input
 295 REM !
1000 REM Cinout;call 703;call 4264
1020 REM cp 255;jr z,Cinout
1040 REM push af;rst 16
1060 REM Cinloop;call 703;call 4264
1080 REM cp 255;jr nz,Cinloop;pop af
1100 REM ret
1110 REM finish
```

Figura 6.2

Effettuare l'output di un valore in un registro è praticamente il processo inverso di quello descritto sopra ma è estremamente più complesso. La figura 6.3 illustra una subroutine per l'output di un valore conservato nell'accumulatore.

```
 10 REM go
 15 REM org 23760
 20 REM !subroutine per visualizzare il numero dal registro A
 25 REM !conserva il valore in A
 30 REM push af
 35 REM !visualizza in alto nello schermo
 40 REM ld a,2
 50 REM call 5633
 55 REM !inizia la routine principale
 60 REM pop af
 70 REM ld b,0
 80 REM Loopa;sub 100
 90 REM jp m,Centinaia
100 REM inc b;!conta le centinaia
110 REM jr Loopa
115 REM !visualizza le centinaia
120 REM Centinaia;add a,100
130 REM ld c,a;!conserva il resto del numero
140 REM ld a,b
150 REM cp 0;!non ci sono centinaia
160 REM jr z,Continua
170 REM add a,48;!trasforma in codice
180 REM rst 16
190 REM ld d,1;!flag acceso per indicare che la cifra delle centina
ia e' stata visualizzata
200 REM Continua;ld b,0
210 REM ld a,c
220 REM Loopb;sub 10
230 REM jp m,Decine
240 REM inc b;!conta le decine
250 REM jr Loopb
255 REM !visualizza le decine
260 REM Decine;add a,10
270 REM ld c,a;!conserva il resto del numero
280 REM ld a,b
290 REM cp 0;!non ci sono decine
300 REM jr nz,Next
310 REM ld a,d
320 REM cp 1;!deve visualizzare 0?
330 REM jp nz,Unita'
340 REM ld a,b
350 REM Next;add a,48;!trasforma in codice
360 REM rst 16
370 REM Unita';ld a,c
```

(continua)

```
380 REM add a,48
390 REM rst 16
400 REM ret
410 REM finish
```

Figura 6.3

6.4 I numeri negativi

Tutti i numeri finora usati erano interi positivi e la routine di input numerico analizzata nell'ultimo paragrafo si occupava solo di numeri positivi. Naturalmente il computer può essere programmato in modo tale da riconoscere se un numero è positivo o negativo. Il modo più semplice per ottenere un numero negativo è quello di impiegare l'istruzione:

NEG

che porterà nell'accumulatore il complemento binario del valore. Se il numero è compreso tra 0 e 127, questa istruzione trasformerà il valore in un numero intero con segno fra 0 e −127, come indicato nel capitolo 2. Ad esempio, se l'accumulatore contiene l'equivalente binario di 45 e viene eseguita l'istruzione NEG, esso contiene l'equivalente dell'intero con segno −45. La figura 6.4 illustra il contenuto dell'accumulatore in sistema binario, prima e dopo l'istruzione NEG.

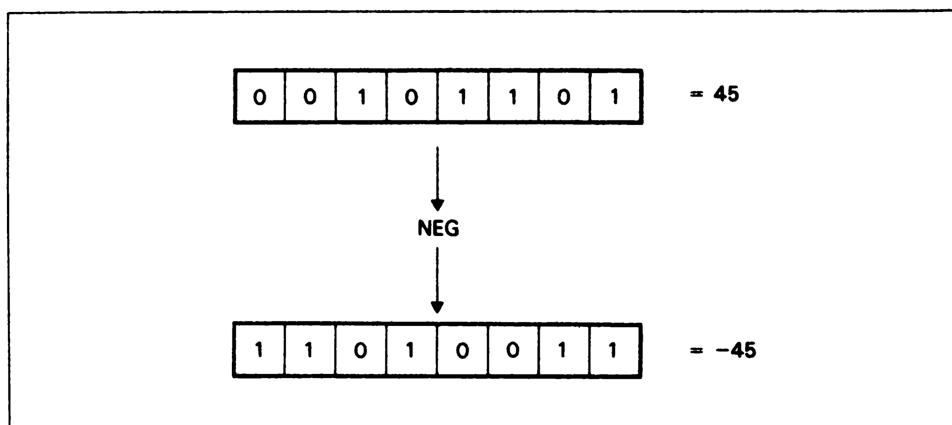


Figura 6.4

Quando si usa la rappresentazione di un intero con segno, il valore in un singolo registro a 8 bit o in una locazione di memoria deve essere compreso tra −128 e +127. Questo limite nell'intervallo dei valori si applica anche ai risultati di tutte le operazioni aritmetiche svolte coi numeri provvisti di segno.
È ora necessario modificare la routine di input numerica per metterla in grado di trattare sia numeri positivi che negativi. Il cambiamento sostanziale concerne il primo carattere che sarà un "−" o un "+" oppure una cifra. Se il primo è un "+" o una cifra, il numero verrà inserito come descritto nel precedente paragrafo; se però il primo carattere è un "−", dopo che le cifre sono state inserite e trasformate in un valore positivo, bisognerà eseguire un'istruzione NEG per trasformarlo in negativo. Provate ad effettuare la modifica della routine come utile esercizio.

6.5 Il riporto e il superamento della capacità numerica

Durante lo svolgimento di operazioni aritmetiche che impiegano dati a 8 bit, tutti i numeri devono trovarsi nell'intervallo compreso tra 0 e 255, se si usano interi senza segno, oppure fra −128 e +127, se con segno. Esamineremo ora il modo di controllare se i risultati si trovano all'interno degli intervalli consentiti.
Osserviamo dapprima l'addizione degli interi senza segno: la figura 6.5 illustra due esempi della somma di numeri ad 8 bit. Nel primo esempio il risultato esatto può essere espresso come un numero a 8 bit e di conseguenza l'operazione è correttamente svolta dal computer; il secondo esempio indica di nuovo l'addizione di due numeri a 8 bit ma il risultato esatto della somma è un numero binario di 9 bit che non può essere con-

```
                00101101        45
              + 10011001     + 153
                ────────     ─────
                11000110       198   Giusto

                10010001       145
              + 10011001     + 153
                ────────     ─────
Riporto = 1     00101010        42   Sbagliato
```

Figura 6.5

	10011001	153	
	− 10010001	− 145	
	00001000	8	Giusto
	10010001	145	
	− 10011001	− 153	
Riporto = 1	11111000	248	Sbagliato

Figura 6.6

tenuto in un registro ad 8 bit, pena la generazione di un risultato errato. In questo modo è possibile verificare se si è ottenuto il risultato corretto: un'addizione, che dà come risultato il valore "1" nel nono bit, lo inserisce in uno dei bit del registro F; questo bit è il flag di riporto e vale uno se un'addizione o sottrazione danno un nono bit, altrimenti vale zero. La figura 6.6 fornisce degli esempi di nono bit, o riporto, risultato di una sottrazione. Il valore in questo bit viene verificato dalla seguente istruzione JP

C = flag di riporto con valore 1

oppure

NC = flag di riporto con valore 0

Il breve segmento di programma presentato qui di seguito ne indica una tipica applicazione:

```
ADD A,B
JP C,ERROR
ERROR;! Inizio di una routine di errore.
```

Le istruzioni di riporto (C) e di non riporto (NC) possono essere usate con quelle di salto relativo (JR).
Allo stesso modo delle istruzioni di addizione e di sottrazione, il flag di riporto può essere modificato da istruzioni di cambiamento e di rotazione, che verranno trattate nel capitolo 10.
Il flag di riporto è molto importante per l'aritmetica del computer, specialmente quando si usano numeri contenuti in più byte; esistono due tipi di istruzioni che consentono il cambiamento diretto del valore del flag

di riporto: SCF, che dà il valore uno al flag di riporto, e CCF, che dà al flag l'opposto del suo valore corrente.
Il registro F comprende anche un bit chiamato flag di superamento della capacità numerica (overflow), impiegato per indicare se il risultato di un'operazione aritmetica è situato al di fuori dell'intervallo concesso ai numeri con segno. Durante l'addizione può aversi l'overflow solo se entrambi i valori sono positivi o negativi, mentre la sottrazione può dare un overflow solo se i due numeri hanno diverso segno. La figura 6.7 elenca degli esempi di addizione e di sottrazione determinanti un overflow.
Il flag di superamento della capacità numerica vale uno se ricorre un overflow, altrimenti vale zero. Il valore è verificato da un'istruzione di salto condizionato. Il flag di overflow possiede inoltre la proprietà di indicare la parità di un byte sul cui concetto non ci soffermeremo poiché l'argomento principale di questo capitolo riguarda l'overflow.
Le istruzioni che verificano il flag di overflow sono:

```
JP PE, label
JP PO, label
```

dove PE significa controllo dell'overflow e PO controllo del non overflow.

	01101010	+ 106	
	+ 01011110	+ 94	
	11001000	− 56	Risposta sbagliata (overflow)
	00011010	+ 26	
	+ 01011110	+ 94	
	01111000	+ 120	Risposta esatta (non c'è overflow)
	10010110	− 106	
	+ 10100010	+ − 94	
Riporto = 1	00111000	+ 56	Risposta sbagliata (overflow)
	01101010	+ 106	
	− 10100010	− − 94	
Riporto = 1	11001000	− 56	Risposta sbagliata (overflow)

Figura 6.7

Non ci sono istruzioni di salto relativo (JR) che verifichino il flag di overflow. La figura 6.8 è una subroutine che preleva due valori dalla memoria e li aggiunge o li sottrae a seconda del codice di carattere contenuto in una terza locazione di memoria; il risultato è poi raccolto in un'altra locazione di memoria. In caso di overflow viene emessa la parola ERROR.
La figura 6.9 illustra un semplice programma in BASIC utilizzabile per controllare questa subroutine.

```
 10 REM go
 20 REM org 23760
 25 REM !alloca la memoria
 30 REM Num1;defb 0
 35 REM Num2;defb 0
 40 REM Sign;defb 0
 45 REM Result;defb 0
 50 REM !inizio del programma
 60 REM ld hl,Num1
 70 REM ld b,(hl);!primo numero
 80 REM inc hl;!punta al secondo numero
 90 REM ld c,(hl);!secondo numero
100 REM ld a,(Sign)
110 REM cp 43;!test +
120 REM jr z,Addit
130 REM cp 45;!test -
140 REM jr z,Subit
150 REM jr Error
155 REM !routine di somma
160 REM Addit;ld a,b
170 REM add a,c
180 REM ld (Result),a
185 REM !controllo di overflow
190 REM jp pe,Error
200 REM ret
205 REM !
210 REM !routine di sottrazione
220 REM Subit;ld a,b
230 REM sub c
240 REM ld (Result),a
245 REM !controllo di overflow
250 REM jp pe,Error
260 REM ret
```

(continua)

```
265 REM !
270 REM !routine Errore
280 REM Error:ld a,2
290 REM call 5633
300 REM ld a,69;!  E
310 REM rst 16
320 REM ld a,114;! r
330 REM rst 16
332 REM ld a,114;! r
335 REM rst 16
340 REM ld a,111;! o
350 REM rst 16
360 REM ld a,114;! r
370 REM rst 16
380 REM ret
390 REM finish
```

Figura 6.8

```
  1 REM 000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000
 10 INPUT "Primo numero?",a
 20 INPUT "Secondo numero?",b
 30 INPUT "A(ddizioni) o S(ottrai)?",x$
 40 POKE 23760,a
 50 POKE 23761,b
 60 IF x$(1)="A"  THEN  POKE  23762,43:  GO TO 90
 70 IF x$(1)="S" THEN  POKE 23762,45: GO TO 90
 80 POKE 23762,0
 90 RANDOMIZE USR 23764
100 PRINT PEEK 23763
```

Figura 6.9

6.6 L'istruzione EQU

L'istruzione EQU è un'altra pseudo-operazione simile alla DEFB trattata nel capitolo precedente. Essa è molto utile al programmatore in Assembler, anche se, una volta tradotto il programma, non produce alcuna

istruzione in codice-macchina. Il modello dell'istruzione per lo ZX Spectrum Machine Code Assembler è:

```
EQU indirizzo nome
```

e il suo scopo è quello di consentire al programmatore di dare una label alle locazioni di memoria anche se al di fuori del programma in codice-macchina. Essa realizza programmi di facile lettura e comprensione ed è particolarmente importante durante la formulazione di un programma di una certa lunghezza suddiviso in diverse sezioni. Il breve programma qui di seguito ne indica un tipico impiego.

```
EQU 5633 Canale
_______
_______
_______
_______
LD A,2
CALL Canale
LD A, 13
RST 16
```

In questo segmento alla locazione il cui indirizzo è 5633, viene dato il nome "Canale" al quale ci si può riferire, piuttosto che impiegare l'indirizzo. L'uso del nome ha lo scopo di rendere più chiare le istruzioni di quanto non sarebbero se si impiegasse l'indirizzo.

6.7 Esercizio

Scrivete un programma con le subroutine analizzate in questo capitolo, che richieda l'inserimento di due numeri dopo aver effettuato il seguente output:

INPUT DEL PRIMO NUMERO
poi INPUT DEL SECONDO NUMERO

Il secondo numero dovrebbe essere positivo, mentre il primo può essere sia positivo che negativo; si moltiplichino i due numeri mediante l'addizione ripetuta. Se ricorre un overflow deve uscire la parola ERROR, in caso contrario il risultato sarà l'output.

Capitolo

I numeri a 16 bit 7

7.1 Coppie di registri

Nello Spectrum, la gran parte dei dati viene manipolata sotto forma di numeri a 8 bit anche se talvolta è più conveniente elaborare dei dati come numeri a 16 bit; in particolare operare a 16 bit quando l'elaborazione riguarda la determinazione di un indirizzo per una locazione di memoria, diviene quasi essenziale. A parte i registri specifici a 16 bit, impiegati per scopi particolari e la cui utilizzazione sarà trattata in seguito, tutti i registri a 8 bit per fini generici si possono usare a coppie per formare i registri a 16 bit BC, DE HL. La coppia di registri HL è sovente impiegata in qualità di accumulatore a 16 bit e solo raramente come due distinti registri ad 8 bit.
Tutti i registri a 16 bit hanno la possibilità di essere caricati direttamente con un valore numerico mediante l'istruzione:

```
LD dd,nn
```

dove nn corrisponde ad un numero variabile tra 0 e 65535 e dd ad uno dei registri a 16 bit BC, DE, HL, IX, IY, SP.
Le sole istruzioni che consentono la trascrizione dei dati da un registro a 16 bit ad un altro sono quelle che usano il registro SP, che sarà oggetto di discussione particolareggiata nel capitolo 8. Le informazioni possono venire trascritte da una coppia di registri ad un'altra considerandoli come registri semplici ad 8 bit; ad esempio, per trascrivere il valore del registro HL al registro BC dovrebbero essere usate le seguenti istruzioni:

LB B,H
LD C,L

Solamente i valori nelle coppie di registri HL e DE possono venire scambiati mediante l'istruzione:

EX DE,HL

7.2 Le informazioni a 16 bit nella memoria

Alcune istruzioni consentono di trasferire dati fra la memoria e le coppie di registri a 16 bit. Tuttavia, poiché ogni locazione di memoria può contenere solo il valore di 8 bit, i valori di 16 bit verranno allocati in due locazioni successive. Quando un valore a 16 bit è spostato da una coppia di registri alla memoria, gli ultimi 8 bit vengono trasferiti alla prima locazione e i primi 8 bit alla seconda. La figura 7.1 illustra tale processo. Lo spostamento dalla memoria ad una coppia di registri si svolge in modo analogo; le istruzioni per realizzare questi movimenti sono:

LD dd,(nn) e LD(nn),dd

dove dd corrisponde ad una delle coppie di registri e nn all'indirizzo della prima delle due locazioni di memoria.

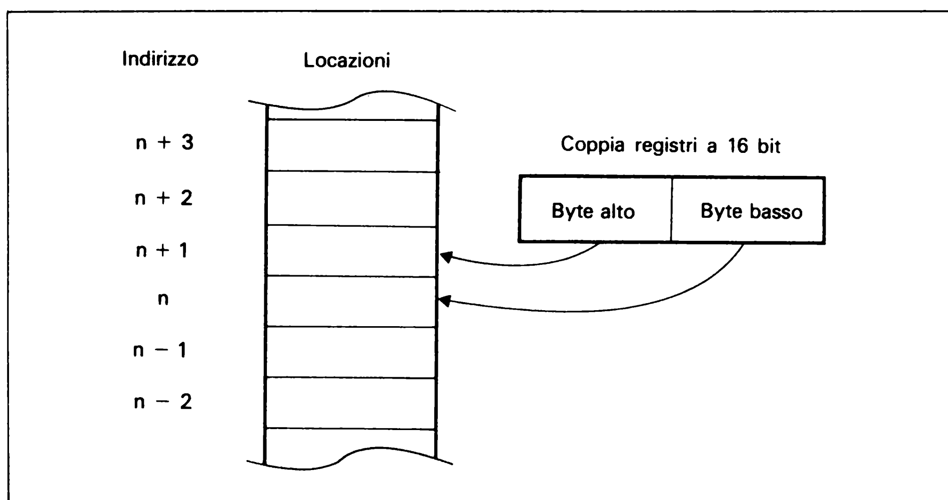


Figura 7.1

7.3 Uso avanzato della tastiera

Nel precedente capitolo si è esaminato come usare la subroutine della ROM per l'inserimento di un codice di carattere nell'accumulatore. Tutti i programmi che si vogliono scrivere in Assembler possono essere scritti semplicemente usando questa subroutine per decodificare qualsiasi input dalla tastiera. Talvolta può nascere il desiderio di impiegare la tastiera in un modo diverso da quello normale, in cui si assegna un codice separato a ciascun tasto. Ad esempio, in un programma di giochi si potrebbe stabilire che tutti i tasti alla sinistra della tastiera spostino il gioco nella medesima direzione, mentre tutti quelli alla destra lo spostino verso destra. Per ottenere questa elasticità, è necessario esaminare più attentamente come il microprocessore acquisisce i dati dalla tastiera. Tutte le informazioni che giungono all'unità centrale di elaborazione, eccetto i dati dalla memoria, vengono inserite tramite una delle istruzioni speciali di input. L'input dalla tastiera usa l'istruzione:

IN A,(n)

Quando si incontra tale istruzione, il valore n, cioè il numero della porta, fornisce l'informazione che segnala al sistema quale periferica, o parte di essa, deve essere esaminata per acquisire i dati che sono poi introdotti nell'accumulatore; questa istruzione inserisce solo un byte di dati alla volta.

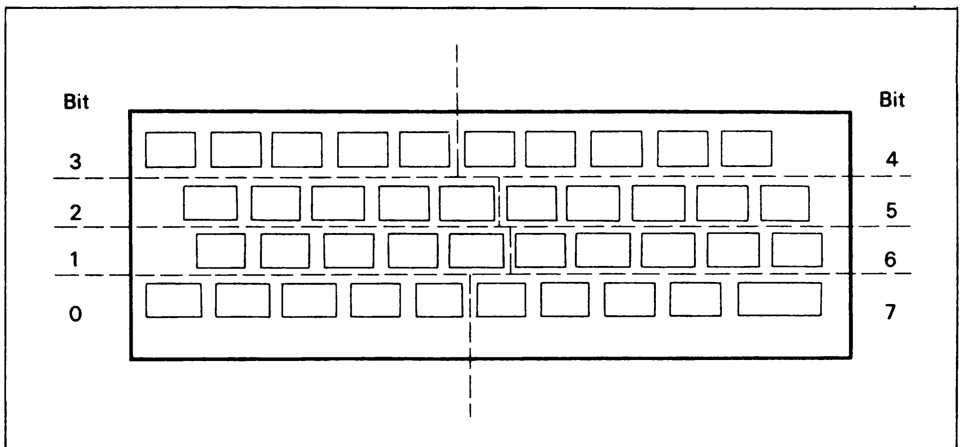


Figura 7.2

Si osservi ora il modo in cui appare la tastiera dello Spectrum al microprocessore e come è possibile scrivere un programma per controllarla. Quando si preme un tasto si producono due segnali; uno si dirige alla porta 254 e si può inserire nel microprocessore mediante l'istruzione IN A,(254). Questo segnale è prodotto trattando la tastiera come otto mezze righe, ciascuna formata da cinque tasti. Premendone uno qualsiasi in una riga di cinque, si pone uguale a zero uno dei bit nel segnale a 8 bit; al contrario, se non viene premuto nessun tasto, tutti i bit vengono posti uguali a uno. La figura 7.2 illustra le mezze righe e il numero del bit assegnato ad ogni gruppo di tasti. È facilmente riscontrabile che la pressione di un tasto alla sinistra della tastiera pone uno dei bit 0, 1, 2 e 3 uguale a zero, mentre un tasto alla destra darà questo valore ai bit da 4 a 7. La figura 7.3 rappresenta un breve programma che stampa un carattere al centro del video e lo sposta a destra o a sinistra in relazione a quale metà della tastiera viene premuta.

```
 10 REM go
 20 REM org 23760
 30 REM Col;defb 15;!posizione iniziale
 35 REM Car;defb 42;!carattere
 37 REM ld a,(Col);ld e,a;!inizia
 40 REM ld a,(Car);ld h,a
 45 REM call Print
 50 REM Esplora;ld d,0;ld bc,65278;!esplora la tastiera
 60 REM Filadopo;inc d;in a,(c)
 70 REM cpl;and 31
 80 REM jr nz,Parte;rlc b;jr c,Filadopo
 90 REM jr Esplora
100 REM Parte;ld a,d;!da che parte e' stato premuto il tasto?
110 REM cp 4;jp p,Destra
120 REM Sinistra;ld l,e;dec e;!sposta a sinistra
130 REM ld a,e;ld (Col),a;cp 1
140 REM jr z,Centro
150 REM call Printer
160 REM ret
170 REM Destra;ld l,e;inc e;!sposta a destra
180 REM ld a,e;ld (Col),a;cp 31
190 REM jr z,Centro
200 REM call Printer
210 REM ret
500 REM Print;ld a,2;!visualizza alla posizione
510 REM push bc;push de;push hl
520 REM call 5633
530 REM pop hl;pop de;pop bc
```

(continua)

```
540 REM ld a,22;rst 16
550 REM ld a,11;rst 16
560 REM ld a,e;rst 16
570 REM ld a,h;rst 16
580 REM ret
600 REM Printer;ld e,1;!cancella la vecchia e visualizza la nuova
610 REM ld h,32
620 REM call Print
630 REM ld a,(Col);ld e,a
640 REM ld h,42
650 REM call Print
660 REM ret
700 REM Centro;ld a,15;ld (Col),a;!rimbalza al centro
710 REM call Printer
720 REM ret
730 REM finish
```

Figura 7.3

7.4 Il suono nel computer

Una delle peculiarità dello Spectrum è la sua capacità di inviare degli output attraverso il suo altoparlante interno. Il suono prodotto mediante la trasmissione di impulsi elettrici all'altoparlante, è realizzabile direttamente dal codice-macchina inviando i dati alla porta collegata con l'altoparlante oppure, più semplicemente, sfruttando la subroutine del suono della ROM.

L'impiego delle porte di output sarà oggetto di discussione in uno dei seguenti capitoli. Questo paragrafo illustra invece l'uso della subroutine della ROM; essa parte dall'indirizzo 949 nella memoria ed è attivata mediante l'istruzione:

```
CALL 949
```

Prima che possa essere usata la subroutine è indispensabile introdurre i valori nelle coppie di registri HL e DE. Il valore nel registro HL controlla l'intervallo di tempo fra gli impulsi trasmessi all'altoparlante, controllando in tal modo l'altezza della nota emessa. La nota emessa è tanto più alta quanto più basso è il valore di HL perché l'intervallo tra gli impulsi è più breve. Il valore nel registro DE controlla la lunghezza della nota e la nota è tanto più lunga quanto maggiore il suo valore.

La figura 7.4 illustra un programma elementare che dà un solo suono. Se si tentano degli esperimenti assegnando diversi valori ai registri DE e

```
 1 REM 0000000000000
10 REM go
20 REM org 23760
30 REM ld hl,650;! tonalita'
40 REM ld de,100;! durata
50 REM call 949; ! routine sonora
60 REM ret
70 REM finish
```

Figura 7.4

HL, si noteranno risultati sorprendenti. Dovreste anche provare un programma che cambi i valori nei registri HL e DE mediante dei loop: verrebbero emessi sicuramente suoni interessanti e forse insoliti.

7.5 I modi di indirizzamento

Quando si formula un programma in BASIC, si scrivono istruzioni come la seguente:

LET A=B+5

senza necessità di precisare se ci si riferisce al valore nella locazione A o all'indirizzo della locazione A. Invece, durante la scrittura di programmi in Assembler, bisogna assicurarsi di specificare dati esatti nelle istruzioni con l'impiego di metodi detti "modi di indirizzamento". Nel corso di questo paragrafo saranno riassunti tutti quelli finora incontrati e verrà spiegato il modo in cui essi indirizzano i dati.
Il microprocessore Z80 possiede 10 diversi modi di indirizzamento e in questo testo ne sono già stati impiegati diversi.
Il *modo diretto* per indirizzare ha il valore corrente dei dati nell'istruzione.
Il *modo via registro* utilizza il nome di un registro in qualità di operando dell'istruzione; i dati vengono conservati in quel registro.
Il *modo esteso* di indirizzare impiega i dati dalla memoria. L'operando dell'istruzione è l'indirizzo della locazione di memoria che contiene i dati. Nell'Assembler l'indirizzamento esteso è indicato mettendo tra parentesi l'operando; questo modo è anche definito *indiretto*.
Il *modo relativo* è impiegato solo per le istruzioni di salto relativo, con o senza condizioni; l'operando per questi tipi di istruzioni è lo spostamento

o il numero delle locazioni di memoria all'istruzione da eseguire se si effettua il salto.
Il *modo implicito* utilizza l'istruzione stessa per indicare i dati da impiegare. Le istruzioni che fanno uso di questo metodo non hanno un operando separato.

Istruzione	Codice-macchina	Indirizzamento
SUB 45	214, 45	Diretto
INC B	4	via Registro
LD A,(32000)	58, 00, 125	Esteso
JR C,LOOP	56, 54	Relativo

Figura 7.5

La figura 7.5 fornisce esempi di questi modi di indirizzamento ed illustra come sono caricati nella memoria. Esistono altri due modi degni di nota. Quello di registro indiretto usa i dati dalla memoria, come il modo esteso, però l'operando nell'istruzione è una coppia di registri che racchiude l'indirizzo della locazione di memoria che contiene dei dati. Ad esempio l'istruzione:

```
LD A,(HL)
```

prende il valore nella coppia HL considerandolo indirizzo di una locazione di memoria e quindi copia il valore di quella stessa locazione di memoria, come è presentato nella figura 7.6. Quando si caricano dei dati nell'accumulatore, può essere usata una qualsiasi delle coppie di registri a 16 bit per contenere l'indirizzo della locazione di memoria, ma si può impiegare solo la coppia HL per caricare uno degli altri registri semplici a 8 bit. È possibile applicare questo metodo alle istruzioni aritmetiche e di salto come le seguenti:

```
ADD A,(HL)
SUB (HL)
JP (HL)
```

Se si opera con istruzioni aritmetiche o di salto, può essere usata solamente la coppia di registri HL per contenere l'indirizzo della locazione di memoria. La figura 7.7 illustra alcuni esempi di questo metodo.
Questo modo di indirizzamento può essere usato sia per il trasferimento

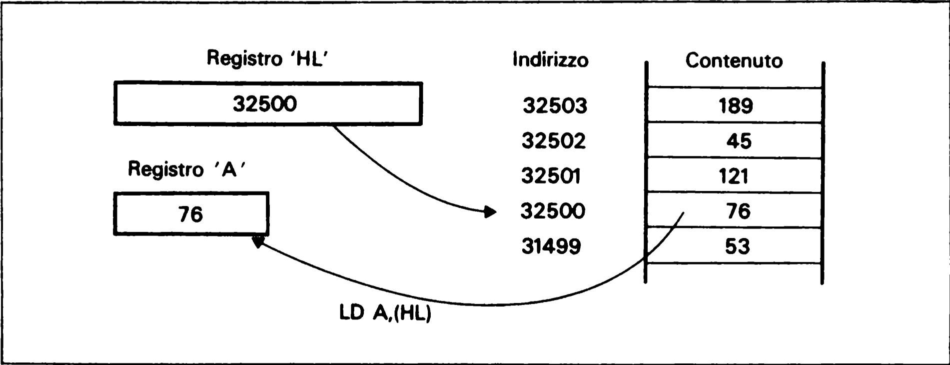


Figura 7.6

Registro 'A'
76
Registro 'HL'
31502
31502
31501
31500
31499
31498
45
121
75
84
21
SUB (HL)
31

31499
31
31502
31501
31500
31499
31498
45
121
75
31
21
LD (HL), A

Figura 7.7

dei dati dalla memoria all'accumulatore o a qualsiasi altro registro a 8 bit, che per trasferire dati dai registri alla memoria. Un esempio è indicato nella figura. La maggiore flessibilità che si ottiene usando la coppia HL è il motivo per cui essa ha la funzione di puntatore principale di memoria.
Il modo di indirizzamento diretto ed esteso è un ampliamento di quello semplicemente diretto: mentre il secondo utilizzava solo i valori a 8 bit, il primo sfrutta quelli a 16 bit e serve per introdurre direttamente un valore in una delle coppie di registri a 16 bit. Ad esempio, l'istruzione:

LD BC,15200

immetterà il valore 15200 nella coppia BC. A ciascuno dei registri a 16 bit BC, DE, HL, SP, IX, IY può essere assegnato un valore. La figura 7.8 rappresenta una sezione di un breve programma che illustra l'impiego dei due modi per indirizzare, analizzati in ultimo.

```
10 REM go
20 REM org 23760
30 REM !modi di indirizzamento
40 REM ld hl,31500;!
50 REM ld a, (hl);!registro indiretto
60 REM ld hl,31520
70 REM sub (hl);!registro indiretto
80 REM ld (hl),a;!registro indiretto
90 REM ld hl,31522
100 REM jp (hl);!registro indiretto
110 REM ret
120 REM finish
```

Figura 7.8

7.6 Esercizio

Scrivete un programma che modifichi i tasti numerici in una tastiera musicale elementare. A ogni tasto deve corrispondere una nota diversa. È quindi necessario fare diversi tentativi per ottenere una gamma di note. Inoltre, premendo e mantenendo premuto un tasto, deve essere emessa

una nota prolungata; si può raggiungere questo risultato anche accorciando la durata di ogni nota e rimandando alla routine di input da tastiera. È però indispensabile modificare la routine di input per eliminare l'eco.

Capitolo

Le iterazioni 8

8.1 I loop

Qualsiasi programma che ripete più di una volta una parte di se stesso, impiega un loop. Probabilmente la caratteristica più importante di un computer qualsiasi consiste proprio nella sua capacità di ripetere le medesime cose più volte senza produrre errori, vale a dire di poter svolgere un loop di programma. È possibile riconoscere differenti tipi di loop in base al metodo usato per determinarne l'uscita.
Il tipo più semplice di loop è quello senza uscita. La figura 8.1 ne illustra

```
 10 REM go
 20 REM org 23760
 30 REM equ 5633 Chanopen
 40 REM ld a,2
 50 REM call Chanopen
 60 REM ld a,31
 70 REM Loop;inc a
 75 REM ld b,a
 80 REM rst 16
 85 REM ld a,b
 90 REM jp Loop
100 REM ret
110 REM finish
```

Figura 8.1

uno semplice che mostra un'immagine sullo schermo. I loop senza uscita non dovrebbero essere normalmente impiegati poiché la loro interruzione richiede lo spegnimento del computer.
Un metodo rapido per fermare un loop è quello di impiegare una verifica di una condizione specifica. Il programma nella figura 8.2 inserisce dei caratteri dalla tastiera e li introduce in locazioni di memoria successive fino a che viene premuto il tasto ENTER. Le istruzioni nel loop sono ripetute fino a che la routine di input immette il codice di carattere 13, cioè quello di ENTER. L'istruzione di confronto seguente alla routine di input darà un risultato di zero quando viene premuto ENTER ed il salto condizionato, JP NZ, Input, verrà ignorato permettendo al programma di continuare all'istruzione "Continue".

```
  10 REM go
  20 REM org 23760
  30 REM ld hl,32499
  40 REM !memorizza i dati a partire dalla cella 32500
  50 REM Input;inc hl
  55 REM push hl
  60 REM call Cinout
  65 REM pop hl
  70 REM ld (hl),a
  80 REM cp 13;!il tasto e' ENTER?
  90 REM jr nz,Input
 100 REM Continue;ret
 110 REM !
1000 REM Cinout;call 703;call 4264
1020 REM cp 255;jr z,Cinout
1040 REM push af;rst 16
1060 REM Cinloop;call 703;call 4264
1080 REM cp 255;jr nz,Cinloop;pop af
1100 REM ret
1110 REM finish
```

Figura 8.2

8.2 I loop a contatore

Uno dei modi più semplici per concludere un loop è quello di compierlo un numero fisso di volte; questa operazione è chiamata loop a contatore: il programma di figura 8.3 ne offre un esempio. Questo programma impiega il registro B per contare il numero di volte in cui viene eseguito il

```
 10 REM go
 20 REM org 23760
 30 REM ld b,5;!ripeti 5 volte
 40 REM ld hl,32500
 50 REM Loop;push hl;push bc;call Cinout
 55 REM pop bc;pop hl
 60 REM sub 48;!trasforma il codice di un numero nel suo valore
 70 REM add a,(hl)
 80 REM ld (hl),a
 90 REM djnz Loop
100 REM call Numout
110 REM ret
520 REM !subroutine per visualizzare il contenuto del registro A
530 REM Numout;push af
540 REM ld a,2
550 REM call 5633
560 REM pop af
570 REM ld b,0
580 REM Loopa;sub 100
590 REM jp m,Centinaia
600 REM inc b
610 REM jr Loopa
620 REM Centinaia;add a,100
630 REM ld c,a
640 REM ld a,b
650 REM cp 0;!ci sono centinaia?
660 REM jr z,Continua
670 REM add a,48;!codifica
680 REM rst 16
690 REM ld d,1
700 REM Continua;ld b,0
710 REM ld a,c
720 REM Loopb;sub 10
730 REM jp m,Decine
740 REM inc b
750 REM jr Loopb
760 REM Decine;add a,10
770 REM ld c,a
780 REM ld a,b
790 REM cp 0;!ci sono decine?
800 REM jr nz,Poi
810 REM ld a,d
820 REM cp 1;!occorre visualizzare 0?
830 REM jp nz,Unita'
840 REM ld a,b
```

(continua)

```
 850 REM Poi;add a,48;
 860 REM rst 16
 870 REM Unita';ld a,c
 880 REM add a,48
 890 REM rst 16
 900 REM ret
1000 REM Cinout;call 703;call 4264
1020 REM cp 255;jr z,Cinout
1040 REM push af;rst 16
1060 REM Cinloop;call 703;call 4264
1080 REM cp 255;jr nz,Cinloop;pop af
1100 REM ret
1110 REM finish
```

Figura 8.3

loop. L'istruzione DJNZ diminuisce di uno il valore nel registro B e causa un salto se il valore non è zero. Quando il valore in B diventa zero, significa che il loop è stato eseguito per il numero di volte richiesto ed il programma passa alla istruzione successiva.
L'istruzione DJNZ è l'abbreviazione di "decrementa e salta se il valore non è zero" e equivale alle due seguenti istruzioni separate:

```
DEC B
JR NZ,label
```

L'istruzione DJNZ può essere impiegata solo con il registro B. Si è soliti sfruttare il registro B a causa dell'istruzione DJNZ nonostante altri registri, o anche valori nella memoria, vengano utilizzati per contare i loop. Il programma nella figura 8.3 è scritto con un valore assegnato nel registro B. È meglio, se possibile, rendere le routine il più generali possibile così da poterle applicare anche ad altri programmi. Il programma sarebbe certamente stato migliore se il valore del contatore del loop fosse stato prelevato da un byte di memoria. Il valore può essere inserito nella memoria mediante un programma principale che utilizza la routine.
Il valore del contatore del loop talvolta è anche usato come dato nel programma. La figura 8.4 illustra un breve programma che calcola la somma di numeri successivi. Si noti che in questo programma l'accumulatore è posto uguale a zero prima di essere usato per contenere la somma: ciò perché il valore nei registri o nelle locazioni di memoria può non essere necessariamente zero se non vengono azzerati dal programmatore.
Quando viene impiegata l'istruzione DJNZ, il loop è eseguito fino a che il suo contatore giunge al valore zero. È spesso più utile usare un contato-

```
 10 REM go
 15 REM org 23760
 20 REM call Cinout
 30 REM sub 48
 40 REM ld b,a
 50 REM ld a,0
 60 REM Loop;add a,b
 70 REM djnz Loop
 80 REM call Numout
 90 REM ret
100 REM !
520 REM !subroutine per visualizzare il contenuto del registro A
530 REM Numout;push af
540 REM ld a,2
550 REM call 5633
560 REM pop af
570 REM ld b,0
580 REM Loopa;sub 100
590 REM jp m,Centinaia
600 REM inc b
610 REM jr Loopa
620 REM Centinaia;add a,100
630 REM ld c,a
640 REM ld a,b
650 REM cp 0;!ci sono centinaia?
660 REM jr z,Continua
670 REM add a,48;!trasforma il valore in codice
680 REM rst 16
690 REM ld d,1
700 REM Continua;ld b,0
710 REM ld a,c
720 REM Loopb;sub 10
730 REM jp m,Decine
740 REM inc b
750 REM jr Loopb
760 REM Decine;add a,10
770 REM ld c,a
780 REM ld a,b
790 REM cp 0;!ci sono decine?
800 REM jr nz,Poi
810 REM ld a,d
820 REM cp 1;!occorre visualizzare lo 0?
830 REM jp nz,Unita'
840 REM ld a,b
```

(continua)

```
850 REM Poi;add a,48
860 REM rst 16
870 REM Unita';ld a,c
880 REM add a,48
890 REM rst 16
900 REM ret
1000 REM Cinout;call 703;call 4264
1020 REM cp 255;jr z,Cinout
1040 REM push af;rst 16
1060 REM Cinloop;call 703;call 4264
1080 REM cp 255;jr nz,Cinloop;pop af
1100 REM ret
1110 REM finish
```

Figura 8.4

re fino al raggiungimento di un valore diverso da zero. In questo caso si usa un'istruzione di confronto per determinare il termine del loop. Tale istruzione può essere usata solo con l'accumulatore, cosicché questo ha la funzione di contatore di loop. La figura 8.5 rappresenta un breve programma che illustra questa particolare applicazione dell'accumulatore.

```
10 REM go
20 REM org 23760
30 REM !loop che usa l'accumulatore come contatore
40 REM ld hl,32500;!puntatore di memoria
50 REM ld a,10;!valore di partenza
60 REM Loop;ld b,(hl)
70 REM inc hl
80 REM ld (hl),b
90 REM add a,b
100 REM cp 104
110 REM jr nz,Loop
120 REM ret
130 REM finish
```

Figura 8.5

8.3 Che cosa è uno stack?

Uno stack (o catasta) è un particolare metodo per raccogliere dati in successive locazioni di memoria. La caratteristica più importante di uno stack consiste nel recupero dei dati nell'ordine contrario a quello in cui erano stati memorizzati; in altre parole la prima voce ricavabile è l'ultima memorizzata. Uno stack è un magazzino LIFO (Last In-First Out) molto utile per la memorizzazione dei dati; esiste un registro speciale nel microprocessore che controlla gli stack, le istruzioni per la memorizzazione e il recupero dati. Gli stack vengono impiegati ampiamente nella ROM dal sistema operativo del computer.
La figura 8.6 illustra la maniera in cui gli stack sono memorizzati nella memoria dello Spectrum: lo stack è allocato capovolto, infatti la sua estremità superiore è posta ad un indirizzo più basso dell'estremità inferiore. Questa disposizione si basa sul fatto che, iniziando lo stack in testa alla memoria, la sua dimensione può indefinitamente aumentare fino alla saturazione della memoria.

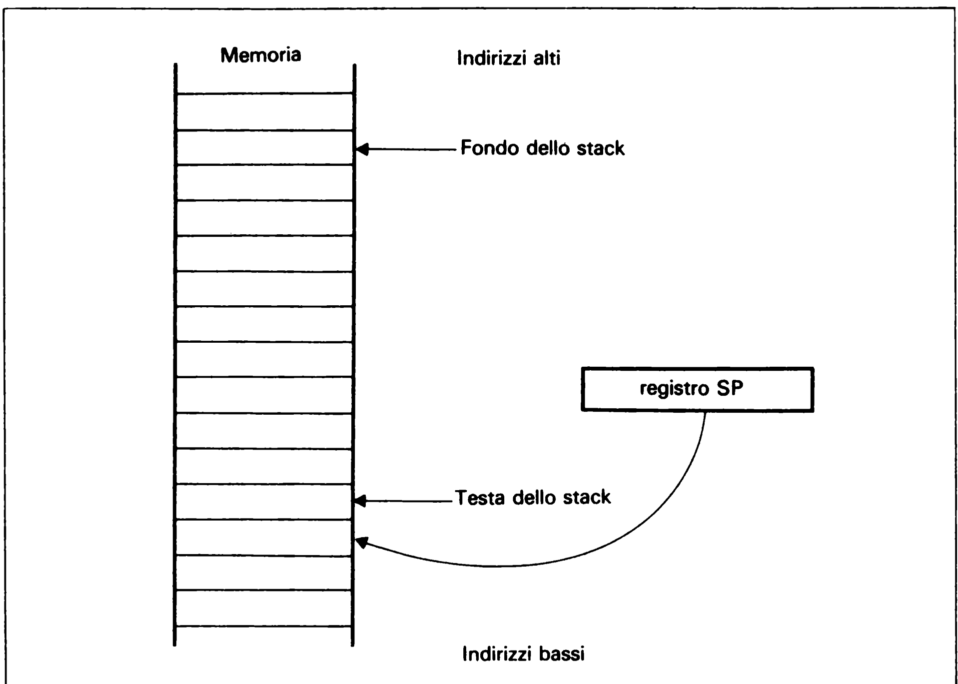


Figura 8.6

Ogni elemento nello stack è lungo due byte; quando si aggiungono o si tolgono dati dallo stack, il valore nell'indice di stack, che punta sempre il primo indirizzo vuoto, aumenta o diminuisce di due.
Gli stack sono impiegati per la memorizzazione temporanea dei dati.

8.4 Gli impieghi degli stack

Uno dei principali impieghi degli stack da parte del sistema operativo del computer è quello di controllare il ritorno da una subroutine al programma principale. Sarà necessario riferirsi alla figura 8.7 insieme alla se-

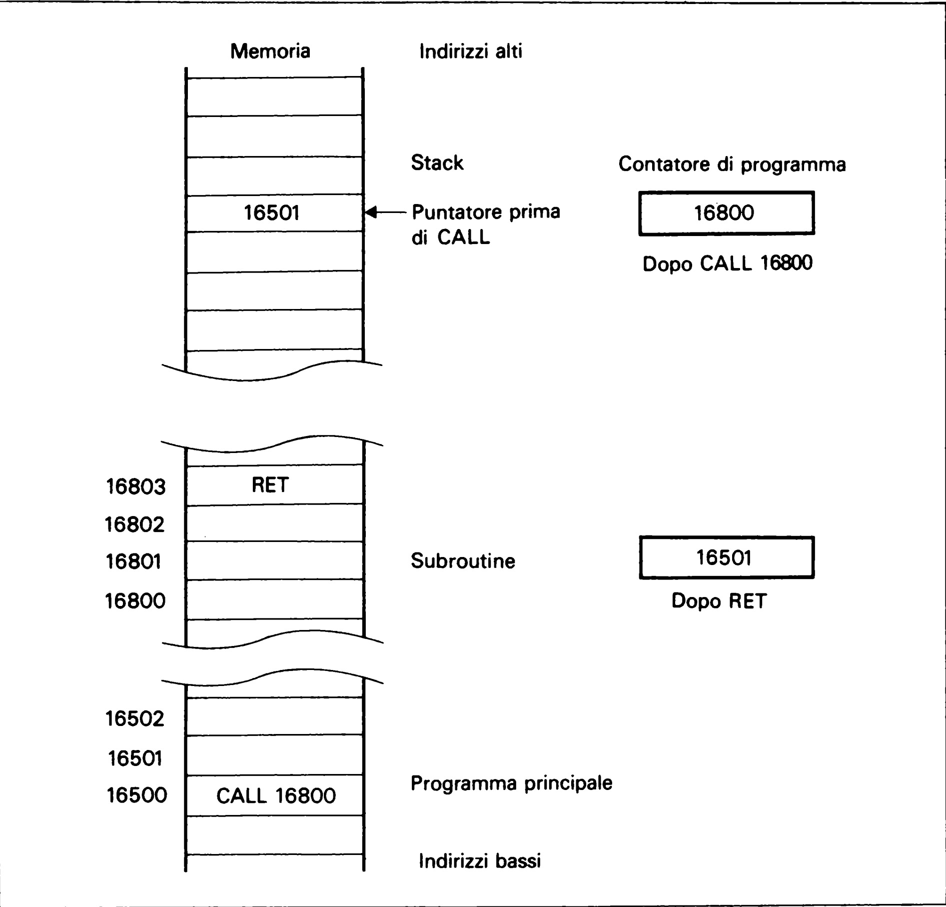


Figura 8.7

guente descrizione. Quando un programma raggiunge un'istruzione CALL, il valore nel contatore del programma, che è l'indirizzo dell'istruzione successiva, è inserito nello stack ed il contatore viene caricato con l'indirizzo della prima istruzione della subroutine; si ha così l'effetto di far eseguire la subroutine. Al termine di questa, l'istruzione di ritorno prende il valore superiore dallo stack e lo introduce nel contatore del programma. La figura illustra una subroutine alla locazione 16800 che ritorna mediante un salto nel programma all'istruzione nella locazione 16501. Prima che venga chiamata la subroutine, l'indice di stack punta la prima locazione vuota. Quando la subroutine raggiunge l'istruzione di ritorno, l'indirizzo dell'istruzione corretta deve trovarsi in cima allo stack; se esso è stato usato nella subroutine, il programmatore deve assicurarsi che lo stesso numero di elementi sia stato aggiunto e tolto dallo stack della subroutine.

Tuttavia è più importante notare l'applicazione degli stack ai programmi. Un impiego comune dello stack consiste nella registrazione dei valori contenuti nei registri del microprocessore mentre si usa una subroutine. A meno che un registro non abbia lo scopo di trasportare le informazioni tra la subroutine ed il programma principale, tutte le subroutine dovrebbero salvare nella memoria i valori di tutti i registri che saranno usati dalla subroutine. Al termine di questa, poco prima di tornare al programma principale, i valori memorizzati possono essere recuperati e sistemati nei registri esatti. Mediante questa tecnica le subroutine possono essere usate da qualsiasi programma, senza per questo preoccuparsi del cambiamento dei valori nei registri durante l'esecuzione delle subroutine. Durante questo impiego dello stack, è importante che i dati siano tolti da esso in ordine inverso a quello in cui sono stati memorizzati.

Esistono diverse istruzioni che consentono al programmatore l'uso degli stack. Nello Spectrum i dati vengono aggiunti allo stack due byte alla volta; questa operazione è nota come "caricamento di dati nello stack", e l'istruzione è la seguente:

```
PUSH rp
```

dove rp rappresenta una qualsiasi delle coppie di registri a 16 bit AF, BC, DE, HL, IX o IY. L'azione di recupero dei dati dallo stack è definita "popping", e l'istruzione è la seguente:

```
POP rp
```

dove rp indica una qualsiasi delle coppie di registri a 16 bit. Le due istruzioni PUSH e POP non solo raccolgono i dati nella zona di memoria occupata dallo stack, ma modificano anche i valori nel registro SP.

La figura 8.8 indica come viene usato lo stack da una subroutine che im-

```
 10 REM go
 20 REM org 23760
 30 REM Subroutine;push bc
 40 REM push de
 50 REM !parte principale della routine
 60 REM !
 70 REM !
 80 REM !
 90 REM !fine dell'elaborazione
100 REM pop de
110 REM pop bc
120 REM ret
130 REM finish
```

Figura 8.8

piega i registri B, C e D per memorizzarvi i valori durante l'esecuzione. Per gran parte delle normali esigenze, il programmatore sfrutterà lo stack fissato dal sistema operativo del computer per la memorizzazione dei dati, ma vorrà talvolta costituirne uno separato per la raccolta dei dati. Ciò è realizzabile introducendo nel registro SP l'indirizzo iniziale di un'area di memoria inutilizzata. Ad esempio, l'istruzione:

LD SP,20000

farà partire uno stack dalla locazione 20000 nella memoria.
Poiché i dati non possono essere spostati direttamente da una coppia di registri a 16 bit ad un'altra, è spesso utile trasferire i dati usando lo stack come memoria di transito.
Esistono anche tre istruzioni che consentono ai dati memorizzati in un registro a 16 bit lo scambio diretto con l'ultimo elemento dello stack. Il modello di questa istruzione è:

EX(SP),rp

dove rp rappresenta uno dei registri a 16 bit HL, IX e IY.

8.5 La visualizzazione dei messaggi

Gran parte dei programmi ad un certo livello della loro elaborazione, hanno bisogno di inviare messaggi allo schermo o alla stampante. Nel BASIC questo è realizzato mediante le istruzioni PRINT o LPRINT, con il messaggio racchiuso tra virgolette. Nell'Assembler, la visualizzazione di un messaggio si svolge in due fasi; la prima consiste nella memorizzazione del messaggio nel programma e nella seconda una sezione del programma invia separatamente il messaggio al display o alla stampante. La prima fase è compiuta mediante una di quelle speciali istruzioni definite pseudo-operazioni; in questo caso l'istruzione è:

DEFS

Essa è solitamente preceduta da una label e seguita dal testo del messaggio; un'istruzione tipica è:

TITLE;DEFS Questo è un programma

```
 10 REM go
 20 REM org 23760
 30 REM Titolo;defs Questo e' un programma
 40 REM push af
 45 REM push bc
 50 REM push hl
 55 REM ld a,2
 60 REM call 5633;!apre il canale
 65 REM !
 70 REM ld b,22;!numero dei caratteri
 75 REM ld hl,Titolo
 80 REM Loop;ld a,(hl);!mette il carattere nel registro A
 85 REM rst 16;!lo visualizza
 90 REM inc hl;!punta al carattere successivo
 95 REM djnz Loop
100 REM pop hl
105 REM pop bc
110 REM pop af
120 REM ret
130 REM finish
```

Figura 8.9

Questa istruzione ha l'effetto di memorizzare il messaggio come una stringa di codici di carattere in locazioni di memoria successive, a partire dalla locazione definita TITLE. Per effettuare l'output del messaggio sul display, la subroutine di stampa, chiamata dall'istruzione RST 16, viene usata all'interno di un loop che conta quanti caratteri devono essere stampati. La figura 8.9 illustra un breve programma che stampa il testo dato nell'istruzione esempio DEFS. Benché questa routine operi in modo perfetto, sarebbe meglio se fosse più generica, così da poter essere impiegata per stampare qualsiasi testo. La figura 8.10 è una versione modifica-

```
 10 REM go
 20 REM org 23760
 30 REM Titolo;defs Questo e' un programma
 40 REM defb 0
 45 REM push af
 50 REM push hl
 55 REM ld a,2
 60 REM call 5633;!apre il canale
 65 REM !
 70 REM !
 75 REM ld hl,Titolo
 80 REM Loop;ld a,(hl);!mette il carattere nel registro A
 85 REM cp 0;!verifica se il testo e' terminato
 90 REM jr z,Fine
 95 REM rst 16;!lo visualizza
100 REM inc hl
105 REM jr Loop
110 REM Fine;pop hl
115 REM pop af
120 REM ret
130 REM finish
```

Figura 8.10

ta della precedente routine; essa preleva l'indirizzo di partenza del testo dal registro HL dove è stato caricato questo indirizzo prima della chiamata della routine. Essa dà luogo all'output dei caratteri da locazioni successive fino a raggiungerne una che contiene il valore zero. Questa subroutine può essere richiamata ogniqualvolta si richiede l'output del testo che deve essere prima memorizzato mediante l'istruzione DEFS seguita immediatamente da un'istruzione DEFB per poter memorizzare il valore zero.

8.6 I loop annidati

Si è finora esaminato il modo in cui è possibile scrivere istruzioni eseguite diverse volte mediante l'inserimento in un loop. Esistono molti programmi dove un blocco di istruzioni, che racchiude un loop, si ripete per diverse volte; in altre parole si ha un loop incluso all'interno di un altro. Questa struttura è chiamata "loop annidato".
Esistono diversi tipi di loop, ciascuno dei quali potrebbe essere utilizzato sia come loop interno che esterno; non c'è alcun limite al genere o al numero di loop annidati. La figura 8.11 ne illustra un esempio.

```
 10 REM go
 20 REM org 23760
 30 REM ld a,2
 35 REM call 5633;!apre il canale
 40 REM !
 45 REM ld a,22;! AT
 50 REM rst 16
 55 REM ld a,11
 60 REM rst 16
 65 REM ld a,0
 70 REM rst 16
 75 REM !
 80 REM ld c,6;!contatore loop esterno
 85 REM Esterno;ld a,23;! TAB
 90 REM rst 16
 95 REM ld a,9
100 REM rst 16
105 REM !
110 REM ld b,8;!contatore loop interno
115 REM ld a,143;!carattere grafico
120 REM ld d;a;!memorizzazione temporanea
130 REM Interno;rst 16
135 REM ld a,d
140 REM djnz Interno
145 REM !
150 REM dec c
155 REM jr nz,Esterno
160 REM ret
170 REM finish
```

Figura 8.11

8.7 Esercizio

Scrivete una subroutine che cancelli lo schermo stampandovi 24 linee, ciascuna formata da 32 spazi. Modificando la precedente subroutine, scrivete un programma che copra l'intero schermo con tutti i caratteri stampabili e li mantenga visualizzati fino a che non viene premuto un qualsiasi tasto.

Capitolo

Il display 9

9.1 Il file del display

Finora tutti gli output sul display sono stati eseguiti tramite la routine di stampa della ROM dello Spectrum. Si è potuto usare questa routine senza considerare come lo Spectrum genera il display; per sfruttarlo al massimo è indispensabile osservarne da vicino il modo.
Lo Spectrum possiede due aree di memoria impiegate nella gestione del display; la seconda di queste è il file degli attributi che controlla i colori, di cui si tratterà nel capitolo 11. La zona principale è il file di display che controlla la forma di ciò che appare sul video. Per stampare qualcosa sul video è necessario caricare in questo file una serie di numeri che definiscono la forma delle immagini sul video.
Lo schermo dello Spectrum ha 24 righe di 32 caratteri e ognuno di questi è creato da una matrice di 8×8 punti: la figura 9.1 illustra alcuni caratteri così prodotti. Ciascuna riga della matrice del carattere viene poi memorizzata come numero di un byte nel file di display. La figura indica anche questo numero, per ciascuna riga di punti nella matrice del carattere, sia in binario che in decimale. L'analisi dei numeri binari mostra come è stato prodotto il numero, mediante l'impiego di un uno per indicare la presenza di un punto e di uno zero per indicarne l'assenza. Ciascun carattere sul video viene memorizzato come otto numeri di un byte ma questi non vengono posti in locazioni di memoria contigue. Nel file di display, il video è memorizzato in tre sezioni di otto righe, perciò la prima parte raggruppa i numeri che compongono le matrici dei caratteri per le prime otto righe del display. Ciascuna sezione è memorizzata come

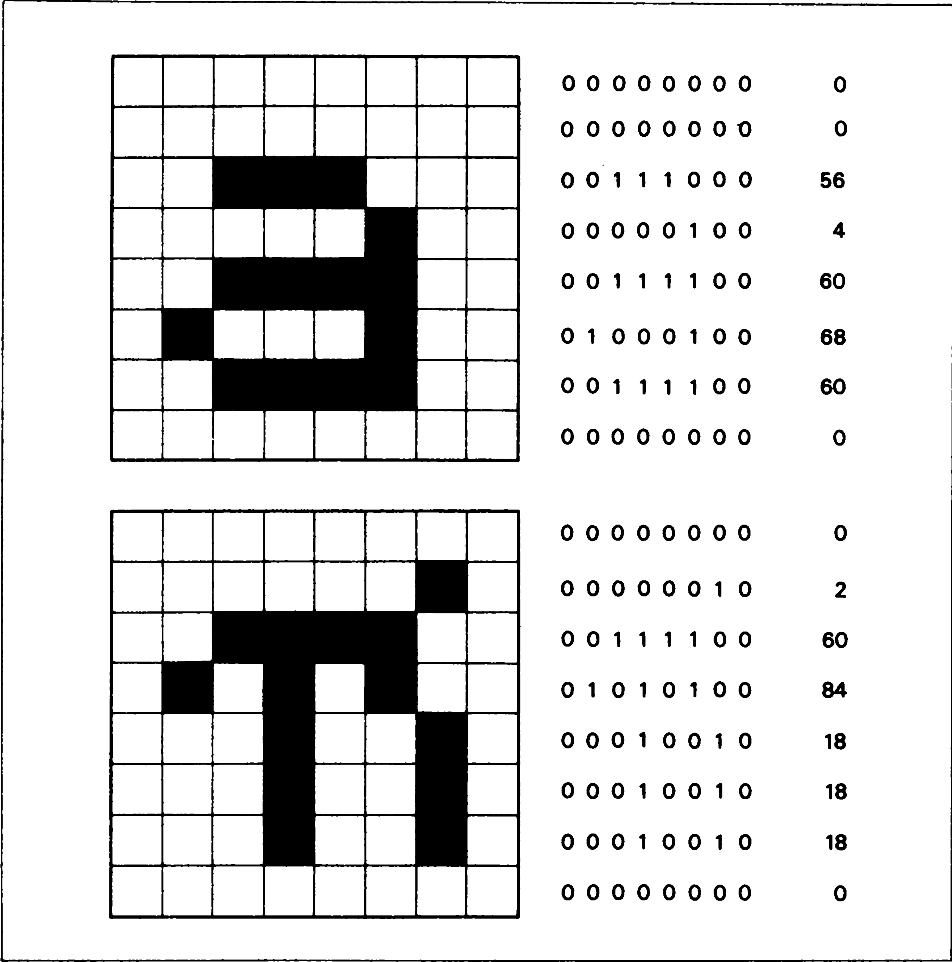


Figura 9.1

mostrato nella figura 9.2. I 32 byte che creano le prime linee delle matrici dei caratteri della prima riga vengono raccolti nelle locazioni di memoria successive, seguiti dai 32 byte ottenuti dalle prime linee dei caratteri della seconda riga. Il processo si ripete fino alla memorizzazione completa delle prime linee delle matrici dei caratteri per le prime otto righe; ora questa operazione viene ripetuta fino alla completa memorizzazione della seconda e delle altre linee delle matrici dei caratteri delle otto righe, come indicato nella figura 9.3.

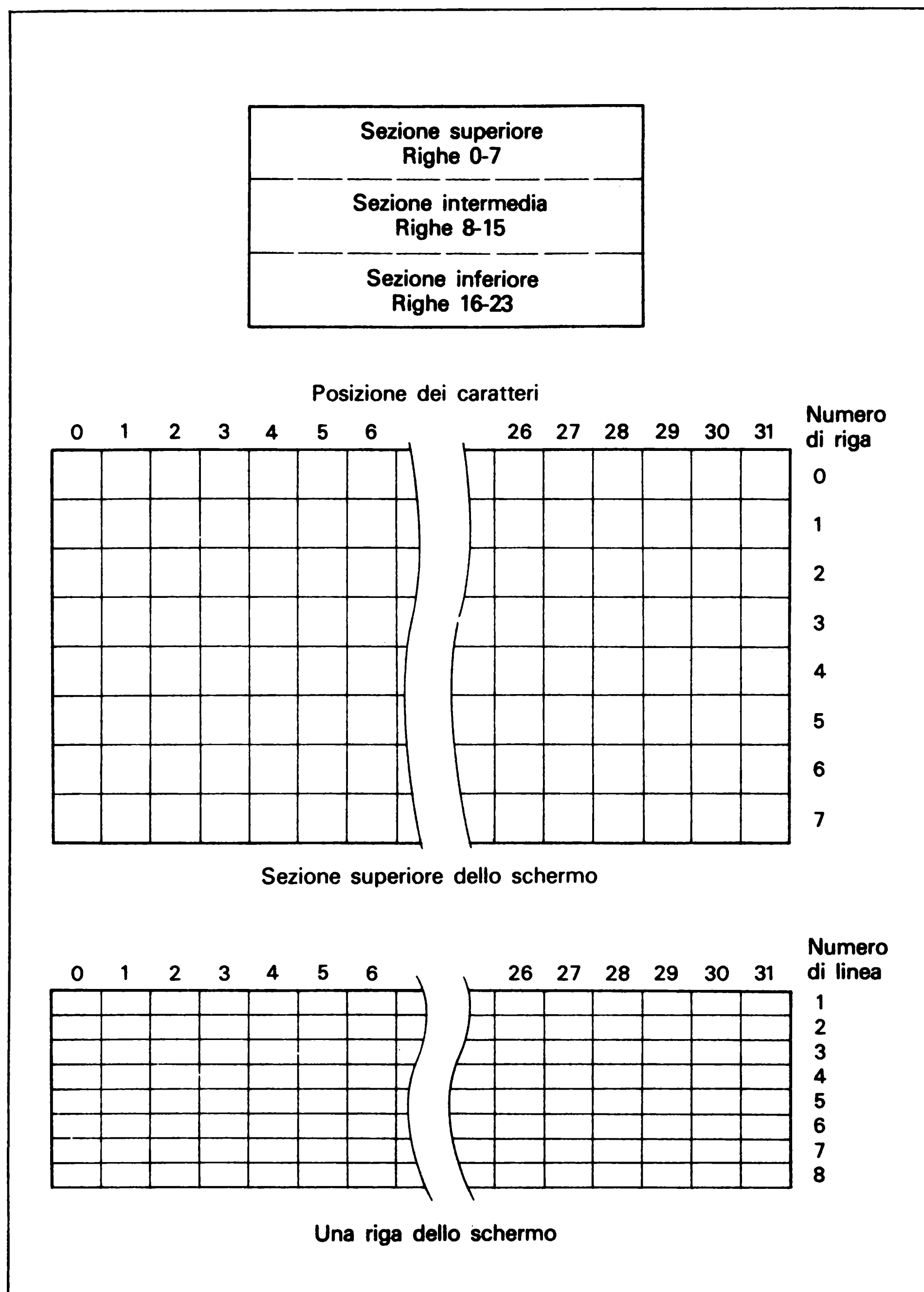

Sezione superiore
Righe 0-7
Sezione intermedia
Righe 8-15
Sezione inferiore
Righe 16-23
Posizione dei caratteri
0 1 2 3 4 5 6 26 27 28 29 30 31
Numero di riga
0
1
2
3
4
5
6
7
Sezione superiore dello schermo
0 1 2 3 4 5 6 26 27 28 29 30 31
Numero di linea
1
2
3
4
5
6
7
8
Una riga dello schermo

Figura 9.2

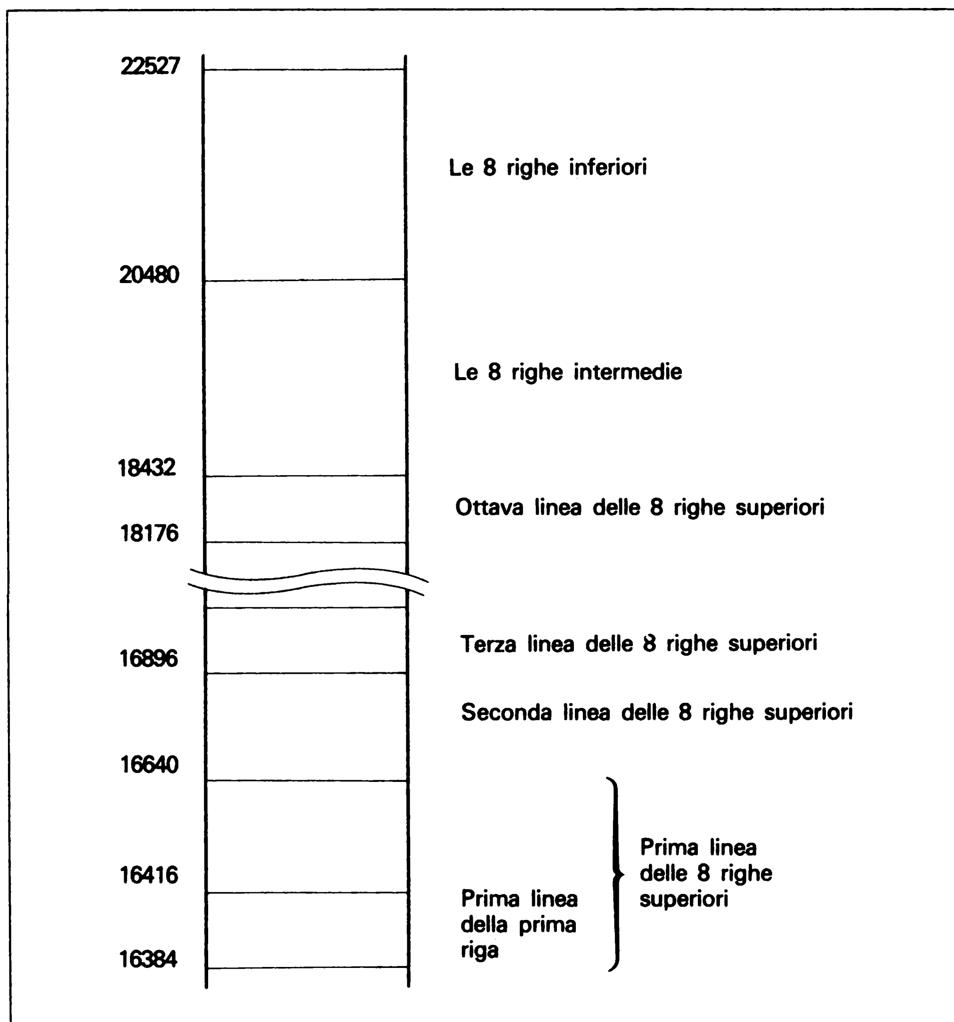


Figura 9.3

9.2 Alla scoperta dei caratteri

Ciascuna linea della matrice dei caratteri è rappresentata da un numero binario a 8 bit, il che significa che ogni linea è memorizzata in una unica locazione di memoria. I numeri che danno origine alle forme di tutti i caratteri standard sono caricati in un blocco di memoria, nella ROM dello

Spectrum, a partire dalla locazione 15616. Ogni carattere è memorizzato come otto numeri in locazioni di memoria successive. Esse sono memorizzate nell'ordine di codice di carattere che comincia con SPACE, che è il codice ASCII 32; esso è poi seguito dagli 8 byte impiegati per creare la forma del carattere il cui codice è 33, e così di seguito. La locazione del primo byte di qualsiasi carattere può essere ricavato dal suo codice mediante la formula:

(codice di carattere−32)*8+15616

I caratteri grafici definiti dall'utente sono memorizzati allo stesso modo, a parte il fatto che vengono raccolti nella RAM invece che nella ROM; questo permette una loro modifica da parte del programmatore. Essi sono generalmente memorizzati in testa alla memoria a partire dalla locazione 32600 nella versione 16K e dalla 65368 in quella da 48K. Uno dei vantaggi derivanti dall'uso dell'Assembler o del linguaggio macchina è che il programmatore può comporre tutti i caratteri che gli sono necessari in qualunque area libera della RAM.

9.3 Spostamenti di blocchi

Tutte le istruzioni in Assembler incontrate sinora impiegano uno o due gruppi di dati, mentre il microprocessore Z80 dello Spectrum ha diverse istruzioni che consentono di usare blocchi completi di memoria.
Il primo gruppo di istruzioni permette il trasferimento dei contenuti di un blocco di memoria in un altro. Poiché il file di display è memorizzato, queste istruzioni possono avere l'effetto di spostare il display o parte di esso. La sezione di programma presentata nella figura 9.4, ad esempio, sposta la decima linea del display alla seconda. L'istruzione importante è la seguente:

LDIR

che significa "carica, incrementa e ripeti".
Prima dell'esecuzione dell'istruzione, la coppia di registri HL deve contenere il primo indirizzo del blocco da spostare, la coppia DE l'indirizzo della prima locazione del blocco di memoria in cui deve essere trasferito e la coppia BC contiene il numero dei byte di dati da muovere; poiché tale numero è incluso in una coppia di registri, il numero massimo di byte trasferibile è 65535. L'azione dell'istruzione LDIR consiste nello spostamento dei contenuti di una locazione di memoria, indicata dalla coppia HL, in quella precisata dalla coppia DE.

```
 10 REM go
 20 REM org 23760
 30 REM equ 16416 Riga2
 40 REM equ 18464 Riga10
 45 REM Temp;defw 0
 50 REM !valori iniziali
 55 REM ld hl,Riga10
 60 REM ld de,Riga2
 65 REM ld c,8;!contatore di linee per riga
 70 REM Loopa;ld b,32;!byte per riga
 75 REM Loopb;ld a,(hl)
 80 REM ld (de),a;!muove un byte
 85 REM inc hl
 90 REM inc de
 95 REM djnz Loopb
100 REM push de;!memorizzazione temporanea
105 REM ld de,224
110 REM add hl,de
115 REM ld (Temp),hl
120 REM pop hl
125 REM add hl,de
130 REM ex de,hl
135 REM ld hl,(Temp)
140 REM dec c
145 REM jr nz,Loopa
150 REM ret
155 REM finish
```

Figura 9.4

Le due coppie di registri HL e DE vengono quindi incrementate di uno mentre la coppia BC diminuisce di uno e tale processo si ripete finché la coppia BC contiene zero.

Usando l'istruzione LDIR si possono avere dei problemi se c'è una sovrapposizione fra i blocchi di memoria. Si consideri il seguente frammento di programma che tenta di spostare il contenuto di un blocco ad un secondo avente indirizzo superiore:

```
LD HL,30000
LD DE,30100
LD BC,500
LDIR
```

Una volta eseguito, i primi cento byte del blocco originale vengono trascritti nel nuovo ma, quando il programma giunge al secondo centinaio, questi sono già stati riscritti. L'effetto di tale programma consiste nel produrre 5 copie del primo centinaio di byte del blocco originale. L'istruzione che risolve questo problema è:

LDDR

che significa "carica, decrementa e ripeti". Essa opera esattamente allo stesso modo dell'istruzione LDIR, con la differenza che a ciascuna fase i contenuti delle coppie di registri HL e DE diminuiscono di uno. Prima dell'esecuzione dell'istruzione le coppie HL e DE devono puntare gli indirizzi più alti nei due blocchi di memoria.
Ci sono inoltre altre due istruzioni che consentono la trascrizione dei blocchi di memoria e sono:

LDI e LDD

Esse assomigliano alle LDIR e LDDR perché trascrivono un byte di dati da un blocco di memoria all'altro; i valori nelle coppie di registri HL e DE vengono cambiati per puntare le successive posizioni nei blocchi, e la coppia BC viene diminuita di uno; diversamente dalle precedenti istruzioni, esse non si ripetono automaticamente fino a che la coppia di registri BC contiene zero. Dopo la trascrizione di ciascun byte deve essere inclusa nel programma una verifica della fine dei blocchi; inoltre è indispensabile effettuare di nuovo un salto all'istruzione LDI o LDD se il ciclo non è completato. Le istruzioni LDI e LDD si usano quando non si conosce la quantità di dati che deve essere trasferita e il programma deve controllare quindi la fine del blocco.
Se si verifica un valore zero nella coppia di registri BC, esso è mostrato non dal flag zero ma dal flag di parità/overflow. Il flag è posto uguale a zero se la coppia BC è zero, altrimenti vale uno; per vedere se il flag è zero, la condizione da verificare è PO, mentre per vedere se è uno la condizione è PE.

9.4 Alcune routine di display

In questo paragrafo saranno presentate alcune semplici routine che consentono al programmatore di riuscire a muovere il display in diversi modi. Probabilmente non si comprenderanno subito tutte le istruzioni, ma non c'è da preoccuparsi perché saranno spiegate fra poco.
La prima routine sposta l'intero video a sinistra; il primo carattere a sini-

```
 10 REM go
 20 REM org 23760
 30 REM equ 16384 Inizio
 40 REM equ 16385 Prossimo
 50 REM !valori iniziali
 60 REM ld de,Inizio
 65 REM ld hl,Prossimo
 70 REM ld b,192
 75 REM !
 80 REM Loop;push bc
 85 REM ld bc,31;!numero di byte
 90 REM ld a,(de);!conserva il primo byte
 95 REM ldir;!sposta una linea
100 REM dec hl
105 REM ld (hl),a
110 REM !linea successiva
115 REM inc hl
120 REM inc hl
125 REM inc de
130 REM pop bc
135 REM djnz Loop
140 REM ret
150 REM finish
```

Figura 9.5

stra di ogni riga si sposta alla fine della precedente e si perde il primo carattere sul video. Il programma è mostrato nella figura 9.5. Anche la routine che segue sposta l'intero schermo ma, questa volta, verso il basso cosicché il display si sposta una riga per volta, scendendo verso la sua estremità inferiore; l'esempio è presentato nella figura 9.6.
Infine una routine utile della ROM dello Spectrum è quella di cancellazione dello schermo ottenuta mediante la parte di programma illustrata nella figura 9.7.

9.5 La cornice del video

Il colore della cornice del video si può cambiare in maniera molto semplice ed ogni volta che lo si reputi necessario tramite l'istruzione:

OUT (254),A

```
 10 REM go
 20 REM org 23760
 35 REM ld hl,22527;!sezione inferiore del video
 40 REM call Spsez
 45 REM call Spriga
 50 REM ld hl,20479;!sezione centrale del video
 55 REM call Spsez
 60 REM call Spriga
 65 REM ld hl,18431;!sezione superiore del video
 70 REM call Spsez
 75 REM call Rigavuota
 80 REM ret;!fine del programma principale
 85 REM !
 90 REM Spsez;ld b,8;!righe
 95 REM Loopa;push bc
100 REM push hl
105 REM ld bc,224
110 REM ld de,32
115 REM scf
120 REM ccf
125 REM sbc hl,de
130 REM pop de
135 REM lddr;!sposta una sezione
200 REM pop bc
205 REM djnz Loopa
210 REM ret
215 REM !
220 REM Spriga;ld b,8;push hl
225 REM ld de,1824;add hl,de
230 REM ex de,hl;pop hl
235 REM Loopb;push bc
240 REM ld bc,32
245 REM lddr;!sposta una riga alla sezione successiva
250 REM ld bc,224
255 REM ex de,hl
260 REM scf
265 REM ccf
270 REM sbc hl,bc
275 REM ex de,hl
280 REM scf
285 REM ccf
290 REM sbc hl,bc
295 REM pop bc;!recupera il contatore
300 REM djnz Loopb
```

(continua)

```
305 REM ret
310 REM !
315 REM Rigavuota;ld b,8
320 REM ld hl,16384
325 REM ld a,0
330 REM Loopd;push bc
335 REM ld b,32
340 REM Loopc;ld (hl),a
345 REM inc hl
350 REM djnz Loopc
355 REM ld de,224
360 REM add hl,de
365 REM pop bc
370 REM djnz Loopd
375 REM ret
380 REM finish
```

Figura 9.6

dove A è il registro che contiene il valore del colore desiderato della cornice. L'istruzione OUT si applica ogniqualvolta il processore centrale del computer invia dei dati all'esterno. A qualsiasi periferica collegata col computer per l'input o per l'output di dati è assegnata una porta provvista di numero; ad esempio, la porta 254 controlla il colore della cornice ma è anche impiegata per il controllo dell'altoparlante; inoltre la stessa serve anche per l'input dalla tastiera. Il cambiamento del colore della cornice è solo temporaneo, a meno che il nuovo valore del colore venga memorizzato nei bit da 3 a 5 della variabile di sistema BORDER. I bit rimanenti di questa variabile di sistema controllano gli attributi della parte inferiore del video. L'appendice G comprende dei particolari di una subroutine per cambiare il colore della cornice.

```
10 REM go
20 REM org 23760
30 REM ld b,24;!cancella 24 linee
40 REM call 3652;!routine della ROM
50 REM ret
60 REM finish
```

Figura 9.7

9.6 Esercizio

Si possono ottenere molti effetti interessanti attraverso la manipolazione diretta del file di display. Scrivete un programma che inverta le otto linee centrali del display. Sarà necessario utilizzare un'altra area di memoria come area di sosta temporanea per parte del display.

Capitolo

La moltiplicazione e la divisione 10

10.1 Le istruzioni di shift

Abbiamo già incontrato dei semplici metodi di moltiplicazione e divisione attraverso l'impiego delle istruzioni ADD e SUB. In questo capitolo esaminiamo le istruzioni e i metodi che consentono una più efficiente elaborazione. Le istruzioni richieste per effettuare le due operazioni sono definite "shift".
Le istruzioni di shift trasferiscono i bit in un registro o in una locazione di memoria di uno spazio, verso sinistra o verso destra. Il microprocessore Z80 dello Spectrum possiede tre diverse istruzioni di shift: SRL, SRA e SLA. La prima ha la struttura:

```
SRL m
```

che significa 'shift logico verso destra', mentre m può essere uno qualsiasi dei registri semplici a 8 bit o il contenuto di una locazione di memoria indicata dalla coppia di registri HL. Questo shift considera il valore da elaborare come un insieme di bit, poiché tutti i bit si spostano di un posto verso destra; si pone uno zero nel bit all'estrema sinistra e il primo bit a destra è posto nel flag di riporto. La figura 10.1 illustra l'operazione di tutte le istruzioni shift.
Il bit estromesso è introdotto nel flag di riporto in tutte le istruzioni shift e le istruzioni di salto condizionato possono poi verificare il valore di questo bit. La seconda delle istruzioni shift ha la seguente struttura:

```
SRA m
```

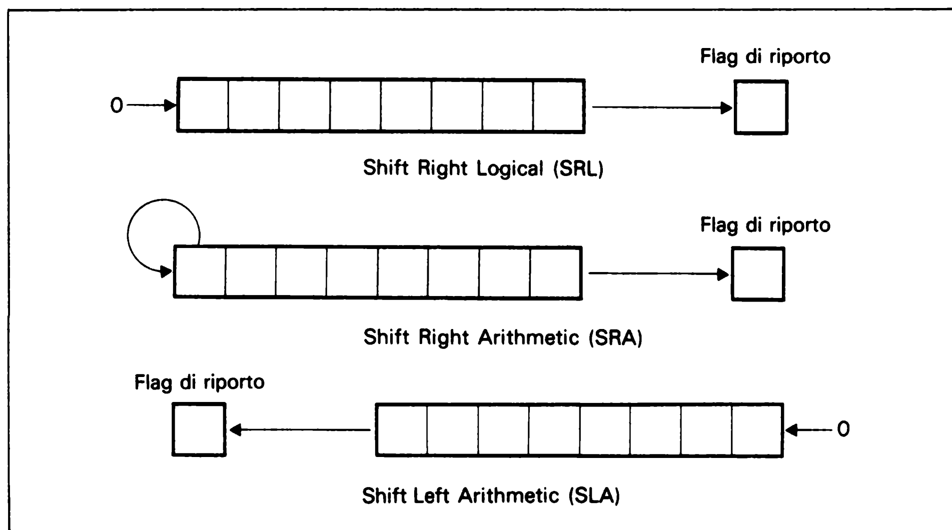


Figura 10.1

che significa "shift aritmetico verso destra", mentre m rappresenta ancora uno qualsiasi dei registri semplici ad 8 bit o una locazione di memoria. L'effetto di questa istruzione è il medesimo dello shift logico verso destra, fatta eccezione per il valore del bit all'estrema sinistra che rimane lo stesso. Uno shift aritmetico verso destra corrisponde alla divisione per due del valore nel registro, o nella locazione di memoria, ponendo l'eventuale resto nel flag di riporto. L'ultima istruzione di shift è la SLA che ha la struttura:

SLA m

che significa "shift aritmetico verso sinistra", mentre m ha sempre il medesimo valore delle istruzioni precedenti. Questo shift sposta tutti i bit verso sinistra e inserisce uno zero nel bit all'estremità destra. L'istruzione di shift verso sinistra ha l'effetto di moltiplicare per due il valore nel registro o nella locazione di memoria.
Un'istruzione di shift logico verso sinistra avrebbe in effetti la stessa funzione di uno aritmetico verso sinistra; non è quindi necessaria un'istruzione separata di shift logico verso sinistra.

10.2 La moltiplicazione

Il metodo impiegato nella moltiplicazione è chiamato "shift e somma" e si basa sugli stessi principi della moltiplicazione. Per esteso può essere illustrato meglio eseguendo passo passo una moltiplicazione tra numeri binari; ma prima consideriamo come opera una moltiplicazione tra decimali:

```
   1537   moltiplicando
  ×2054   moltiplicatore
  -----
   6148   ×4
  7685    ×5
 0000     ×0
3074      ×2
-------
3156998   prodotto
```

La moltiplicazione per esteso nel sistema binario è identica a quella nel decimale. Bisogna sapere come moltiplicare per uno e per zero; dall'esempio potete vedere che è facile:

```
     10101
    ×11001
  --------
     10101   ×1
    00000    ×0
   00000     ×0
  10101      ×1
 10101       ×1
----------
1000001101   prodotto
```

Il metodo "shift e somma" può essere definito così: iniziando dalla destra del moltiplicatore si sommi il moltiplicando al prodotto se il bit del moltiplicatore è uno; si sposti il moltiplicando di un bit a sinistra e si ripeta per il successivo bit del moltiplicatore fino al termine di questo.
La figura 10.2 illustra un programma nel quale si moltiplicano i contenuti dei registri D e E per lasciare il prodotto nell'accumulatore. Il metodo per la divisione si basa su quello per la moltiplicazione e assomiglia molto allo "shift e somma" ma è definito "shift e sottrai".

```
10 REM go
20 REM org 23760
25 REM Mdo;defb 8
27 REM Mre;defb 9
30 REM !moltiplicazione
32 REM ld a,(Mdo);ld d,a;ld a,(Mre);ld e,a
35 REM ld a,0
40 REM ld b,7
45 REM Loop;srl e
50 REM !verifica se il bit vale uno
55 REM jp nc,Prossimo
60 REM add a,d;!lo somma al prodotto
65 REM Prossimo;sla d
70 REM djnz Loop
72 REM ld c,a
75 REM ret
80 REM finish
```

Figura 10.2

10.3 Le rotazioni

Un gruppo di istruzioni, molto simili nell'operare a quelle di shift, è quello delle istruzioni di rotazione. La principale differenza fra le due consiste nel fatto che nella rotazione il bit estromesso è reintrodotto all'altra estremità. Come nello shift, le istruzioni di rotazione fanno spostare i bit di un posto verso sinistra o verso destra.
Alcune di esse utilizzano l'accumulatore come parte dell'istruzione e non come operando separato; queste istruzioni occupano un byte, mentre per le altre, come per le istruzioni di shift, è indispensabile che sia precisato il registro o la locazione di memoria. Queste istruzioni sono lunghe due byte.
Le rotazioni dell'accumulatore possono avvenire sia verso destra che verso sinistra e possono comprendere o no il flag di riporto nella rotazione. Le quattro istruzioni sono:

RLA, RRA, RLCA e RRCA

il cui effetto è indicato nella figura 10.3.
Le istruzioni di rotazione per gli altri registri ad 8 bit o per i contenuti di una locazione di memoria, indicate dalla coppia di registri HL, sono:

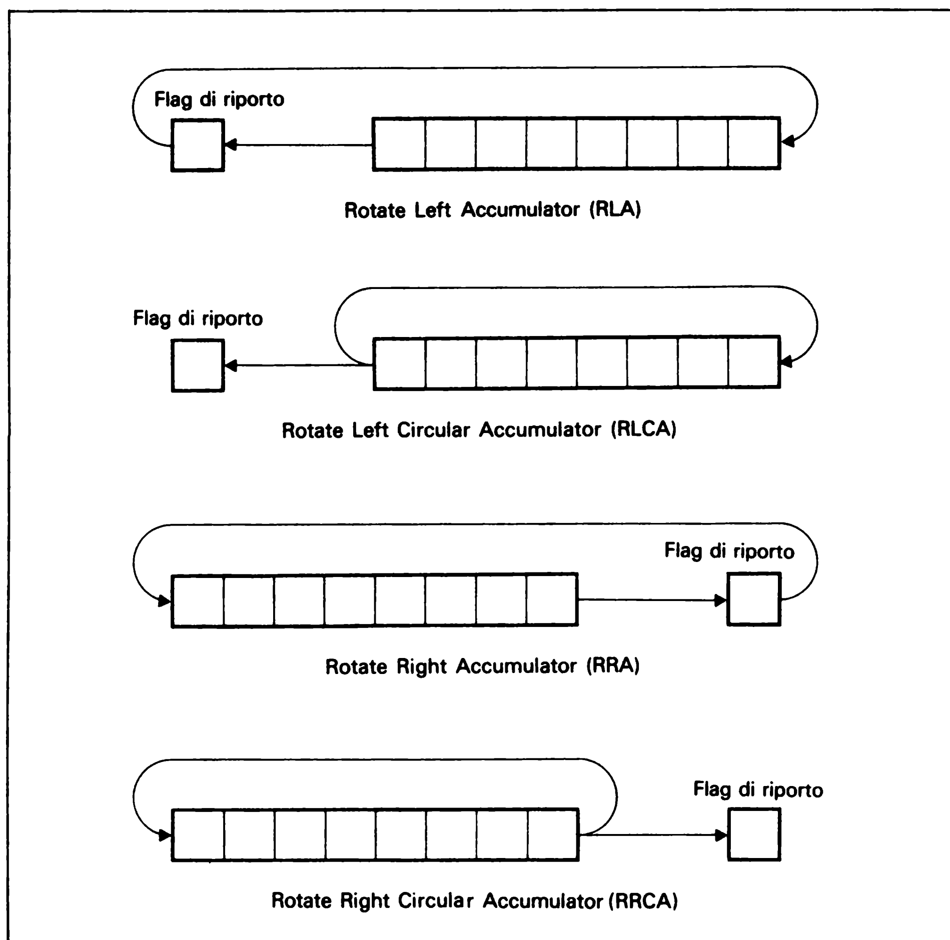


Figura 10.3

RL m	rotazione verso sinistra attraverso il flag di riporto
RR m	rotazione verso destra attraverso il flag di riporto
RLC m	rotazione circolare verso sinistra non attraverso il flag di riporto
RRC m	rotazione circolare verso destra non attraverso il flag di riporto

dove m corrisponde ad uno qualsiasi dei registri ad 8 bit, tranne l'accumulatore, o ad un valore di memoria indicato dalla coppia HL. L'effetto di queste istruzioni è identico a quello indicato per le corrispondenti rotazioni dell'accumulatore.

10.4 Esercizio

Il programma della moltiplicazione illustrato in precedenza riguarda solo dei numeri positivi e dovrebbe essere relativamente semplice estenderlo a quelli con segno. Un metodo consiste nel moltiplicare prima i valori positivi dei due numeri e quindi applicare le regole di moltiplicazione per determinarne il segno. La regola dice che segni uguali danno risultato positivo, mentre segni diversi danno un valore negativo; bisogna ricordare che un numero con segno che ha un uno nell'estremo bit a sinistra è negativo e il suo valore positivo si ricava tramite l'istruzione NEG.
Scrivete una subroutine che esegua una divisione per dieci e usatela poi per formularne un'altra che visualizzi il valore contenuto nell'accumulatore come un numero compreso tra −128 e +127. Tutti i numeri negativi dovrebbero essere preceduti da un segno '−'.
Utilizzando la precedente subroutine ed il programma di moltiplicazione modificato, scrivete un programma che acquisisca due numeri con segno e calcoli il loro prodotto; un display tipico potrebbe essere:

```
4*-6=-24
7*8=56
```

Nel caso desideriate formulare un programma simile per la divisione, poiché si opera con interi, dovrete tener conto del problema dell'eventuale resto e della sua visualizzazione.

Capitolo

Un po' di logica 11

11.1 Le operazioni dei bit

Tutte le istruzioni finora incontrate impiegavano i dati in byte e non si è parlato di nessuna istruzione che usasse meno di 8 bit di dati. Saranno ora prese in esame alcune istruzioni che usano un bit per volta. Per prima cosa è necessario numerare i bit in un registro, o in una locazione di memoria, così da potersi riferire a questi. La figura 11.1 indica che i bit sono numerati da destra a sinistra, a partire dallo zero. Non c'è una ragione particolare per questa convenzione; è standard per tutti i computer. Tutte le istruzioni di bit agiscono su uno dei registri ad 8 bit oppure sui contenuti di una locazione di memoria indicata dalla coppia di registri HL. La prima delle istruzioni ha la struttura:

BIT n,m

dove n è il numero del bit e m un registro a 8 bit o una locazione di me-

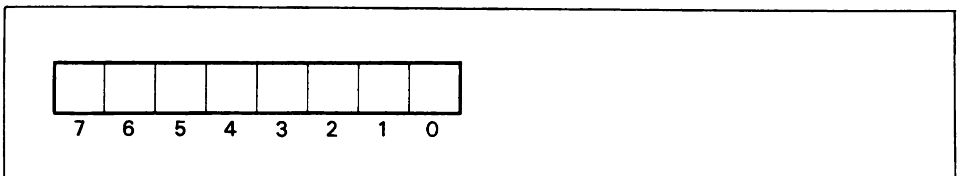


Figura 11.1

moria. Il fine di questa istruzione è quello di verificare il valore nel bit stabilito e di porre il flag di zero conformemente al risultato. Poiché l'istruzione di bit è sempre seguita da un'istruzione di salto condizionato, un frammento di programma tipico potrebbe essere:

```
BIT 4, D
JP Z,NEXTPART
```

Se il registro D contiene il valore 45 (00101101 binario), il bit 4 (si ricordi di contare dal bit zero all'estrema destra) è uguale a zero e il programma salterà all'istruzione definita NEXTPART.

```
 10 REM go
 20 REM org 23760
 30 REM equ 16384 Dispfile
 40 REM !
 45 REM ld hl,Dispfile
 50 REM ld c,192
 55 REM !programma principale
 60 REM Filadopo;ld a,(hl)
 65 REM sla a;!muove il primo byte
 70 REM ld (hl),a
 75 REM !il resto della riga
 80 REM ld b,31
 85 REM Spostalinea;inc hl
 90 REM ld a,(hl)
 95 REM sla a;!fa scorrere un byte a sinistra
100 REM ld (hl),a
105 REM !verifica se il primo bit vale uno
110 REM jp nc,Noriporto
115 REM dec hl;!byte precedente
120 REM ld a,(hl)
125 REM set 0,a
130 REM ld (hl),a
135 REM inc hl
140 REM Noriporto;djnz Spostalinea
145 REM inc hl
150 REM dec c
155 REM jr nz,Filadopo
160 REM ret
170 REM finish
```

Figura 11.2

Nel file di display tutti i punti che compongono la videata sono memorizzati in forma di bit e l'istruzione BIT serve per verificare se un punto particolare sullo schermo è acceso o spento.
Le altre istruzioni che agiscono su bit particolari sono usate per porre il valore del bit a uno o a zero. L'istruzione SET dà il valore uno ad un bit specificato mentre RES dà il valore 0 a un bit. La struttura delle due istruzioni è:

SET n,m   e   RES n,m

dove n corrisponde al numero del bit, mentre m è il registro o la locazione di memoria.
Ora che si è in grado di operare con dei singoli bit è possibile dare uno sguardo alla routine che trasla lateralmente il contenuto dello schermo un bit alla volta. La figura 11.2 illustra una routine che fa traslare il contenuto del video a sinistra e dovreste riuscire a modificarla per ottenerne una che lo fa verso destra.

11.2 Le istruzioni logiche

L'Assembler dello Spectrum possiede un insieme di istruzioni che consente la combinazione dei bit nell'accumulatore con quelli in un registro a 8 bit o in una locazione di memoria, usando le regole degli operatori logici.
Questi operano su bit corrispondenti nell'accumulatore e nell'operando. Gli operatori logici di base sono AND, OR, XOR e NOT. Le norme che regolano la combinazione dei bit sono elencate nella figura 11.3. A parte l'operatore NOT, tutte le operazioni combinano delle coppie corrispondenti di bit e danno come risultato un bit per ciascuna coppia. Le istruzioni logiche sono:

AND   OR   XOR

e la struttura delle istruzioni è:

AND p

dove p corrisponde ad un valore a 8 bit sia esso un registro a 8 bit che il valore in una locazione di memoria indicata dalla coppia di registri HL. La Figura 11.4 fornisce alcuni esempi circa il funzionamento delle istruzioni logiche.

Operatore AND

0 AND 0 = 0
0 AND 1 = 0
1 AND 0 = 0
1 AND 1 = 1

Operatore OR

0 OR 0 = 0
0 OR 1 = 1
1 OR 0 = 1
1 OR 1 = 1

Operatore XOR

0 XOR 0 = 0
0 XOR 1 = 1
1 XOR 0 = 1
1 XOR 1 = 0

Operatore NOT

NOT 0 = 1
NOT 1 = 0

Figura 11.3

Contenuto di A	01011100
Contenuto di B	11001100
Contenuto di A dopo AND B	00001100
Flag di segno = 0	Flag di zero = 0
Contenuto di A	01011100
Contenuto di B	11001100
Contenuto di A dopo OR B	11011100
Flag di segno = 1	Flag di zero = 0
Contenuto di A	01011100
Contenuto di B	11001100
Contenuto di A dopo XOR B	10010000
Flag di segno = 1	Flag di zero = 0
Contenuto di A	01011100
Contenuto di A dopo CPL	10100011

Figura 11.4

Quando un'istruzione logica è terminata, i flag di segno e di zero sono posti in conformità al valore rimasto nell'accumulatore. L'operazione NOT è eseguita mediante l'istruzione CPL; essa opera solo sull'accumula-

tore e ha l'effetto di modificare il valore di tutti i bit nell'accumulatore, infatti tutti gli uno diventano zero e tutti gli zero diventano uno.

11.3 I dati compattati

Il programmatore dovrebbe sempre cercare di minimizzare la quantità di memoria utilizzata dai suoi programmi e dai suoi dati, ottenendo così più spazio per altri programmi o per una maggiore quantità di dati. Mi è stato mostrato recentemente il programma di un elenco di indirizzi per il 16K ZX81 che forniva tutte le prestazioni richieste ad un programma di questo genere ma presentava un grande svantaggio: era troppo ingombrante. Una volta memorizzato il programma, rimaneva spazio solamente per 10 o 12 nomi ed indirizzi.
L'impiego dell'Assembler aiuta certamente a ridurre la dimensione del programma, ma come si può fare per diminuire la quantità di spazio occupata dai dati? Sovente i dati utilizzati variano in un intervallo molto inferiore a quello che può essere contenuto in ogni locazione di memoria, dove è possibile accogliere un numero variabile da 0 a 255 oppure da −128 a +127. Si supponga ora che una delle voci dei dati sia l'età di una persona; ovviamente non saranno necessari i numeri negativi ma anche l'intervallo da 0 a 255 è molto più ampio del necessario. Quindi se si useranno solo sette degli otto bit per l'età, sarà possibile usufruire di un intervallo compreso tra 0 e 127, più che sufficiente e rimarrà così l'ottavo bit utilizzabile per raccogliere qualche altra informazione. Infatti in un bit è possibile accogliere tutte le informazioni aventi solo due possibili risposte, come ad esempio il sesso di una persona o se è un cliente o un fornitore. Tutte queste voci di dati possono venire memorizzate in qualità di ottavo bit nella locazione di memoria che contiene l'età dell'individuo. Questo processo è definito *compattamento dei dati*.
La figura 11.5 illustra un tipico esempio estratto da un elenco di apparte-

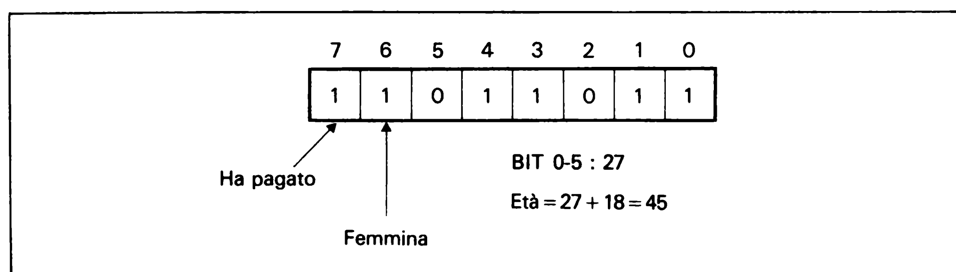


Figura 11.5

nenti ad un club dove, poiché l'intervallo dell'età dei membri è compreso tra i 18 e i 65, i bit da 0 a 5 raccolgono l'età come numero di anni oltre i 18, il bit 6 comprende il sesso dei membri mentre il bit 7 indica se è stata versata oppure no la quota associativa al club. L'esempio illustra il caso di una donna di 45 anni che ha pagato la quota. In un club di 1000 soci grazie a questo metodo verranno impiegati solo 1000 byte di memoria invece dei 3000 che sarebbero stati necessari se ciascuna voce di dati fosse stata memorizzata in una locazione di memoria separata.

11.4 Il compattamento e la separazione dei dati

Si è trattato dei principi di compattamento di dati in un byte; si vedrà ora come vengono eseguite le operazioni di compattamento e separazione. In genere la prima operazione richiede le istruzioni di shift e OR,

```
10 REM go
20 REM org 23818
25 REM !compatta i dati
27 REM Eta;defb 45
28 REM Ses;defb 1
29 REM Pag;defb 1
30 REM Voce;defb 0
31 REM ld a,(Eta);ld b,a
32 REM ld a,(Ses);ld c,a
33 REM ld a,(Pag);ld d,a
35 REM ld a,b
40 REM sub 18;!l'eta' nei bit 0-5
45 REM !
50 REM ld b,6
55 REM Sesso;sla c;!il sesso nel bit 6
60 REM djnz Sesso
65 REM or c;!aggiunge il sesso ad A
70 REM !
75 REM ld b,7
80 REM Pagato;sla d;!pagamento nel bit 7
85 REM djnz Pagato
90 REM or d
95 REM !
100 REM ld (Voce),a;!in memoria
105 REM ret
110 REM finish
```

Figura 11.6

mentre per la seconda sono necessarie le istruzioni di shift e AND.
La figura 11.6 presenta un programma di compattamento dati nella locazione di memoria definita VOCE che impiega le regole fornite nel paragrafo precedente. All'inizio del programma il registro B contiene l'età dell'individuo, il registro C il sesso e il D se ha pagato o no. La figura 11.7 è invece il programma per la separazione dei dati ed inverte la procedura di quello precedente.

```
 10 REM go
 20 REM org 23818
 22 REM Eta;defb 0
 23 REM Ses;defb 0
 24 REM Pag;defb 0
 25 REM Voce;defb 219
 27 REM org 23855
 30 REM !suddivide i dati compattati in 'Voce'
 35 REM ld a,(Voce)
 40 REM and $80;!10000000B
 45 REM ld d,a
 50 REM ld b,7
 55 REM Pagatob;srl d;!al bit 0
 60 REM djnz Pagatob
 65 REM ld a,d;ld (Pag),a
 70 REM ld a,(Voce)
 75 REM and $40;!01000000B
 80 REM ld c,a
 85 REM ld b,6
 90 REM Sessob;srl c;!al bit 0
 95 REM djnz Sessob
100 REM ld a,c;ld (Ses),a
105 REM ld a,(Voce)
110 REM and $3F;!00111111B
115 REM add a,18;!calcola l'eta' esatta
120 REM ld b,a
125 REM ld (Eta),a
130 REM ret
140 REM finish
```

Figura 11.7

11.5 Il file degli attributi

Come già detto, le forme visualizzate sul video vengono memorizzate come singoli punti nel file del display, ma lo Spectrum è un computer a colori ed esiste quindi un'ulteriore zona di memoria che serve a memorizzare i particolari del colore: questa è definita "il file degli attributi". Esso consiste in una locazione di memoria per ogni posizione di carattere sul video, memorizzate nel seguente ordine: 32 locazioni per la prima riga, poi 32 per la seconda riga e via di seguito per tutte le 24 righe. Ciascuna locazione nel file degli attributi contiene quattro dati diversi compattati. I bit da 0 a 2 contengono il codice per il colore del carattere, i bit da 3 a 5 lo sfondo, il bit 6 vale 1 per luminoso e 0 per normale, e il bit 7 vale 1 per lampeggiante e 0 per fisso.
La figura 11.8 mostra come può venire usato il file degli attributi per modificare le caratteristiche di colore del video.

```
10 REM go
20 REM org 23760
24 REM equ 768 Bytes
26 REM equ 22528 File
30 REM !cambio colori nel file di attributo
35 REM !colore del carattere
40 REM call Cinout
45 REM sub 48;!dal codice al valore numèrico
50 REM ld b,a
52 REM push bc
55 REM !
57 REM !colore dello sfondo
60 REM call Cinout
65 REM sub 48
67 REM pop bc
70 REM ld c,a
72 REM push bc
75 REM !
77 REM !doppia luminosita'?
80 REM call Cinout
85 REM sub 48
90 REM ld d,a
92 REM push de
95 REM !
98 REM !lampeggiante?
100 REM call Cinout
105 REM sub 48
```

(continua)

```
 107 REM pop de
 110 REM ld e,a
 112 REM pop bc
 115 REM !
 120 REM !compatta i dati in A
 125 REM xor a;!azzera A
 130 REM ld a,b;!il colore del carattere nei bit 0-2
 135 REM ld b,3
 140 REM Paper;sla c
 145 REM djnz Paper
 150 REM or c;!il colore dello sfondo nei bit 3-5
 155 REM !
 160 REM ld b,6
 165 REM Bright;sla d
 170 REM djnz Bright
 175 REM or d;!luminosita' nel bit 6
 180 REM !
 185 REM ld b,7
 190 REM Flash;sla e
 195 REM djnz Flash
 200 REM or e;!lampeggio nel bit 7
 205 REM !
 210 REM !carica nel file di attributo
 215 REM ld bc,Bytes
 220 REM ld hl,File
 225 REM push hl
 230 REM pop de
 235 REM inc de
 240 REM ld (hl),a
 245 REM ldir
 250 REM ret
 255 REM !
1000 REM Cinout;call 703;call 4264
1020 REM cp 255;jr z,Cinout
1040 REM push af;rst 16
1060 REM Cinloop;call 703;call 4264
1080 REM cp 255;jr nz,Cinloop;pop af
1100 REM ret
1110 REM finish
```

Figura 11.8

11.6 Esercizio

Scrivete una subroutine che mostri sul video, in binario, il valore contenuto nell'accumulatore. Considerate un bit per volta e visualizzate il codice carattere corrispondente a 1 o a 0. Scrivete poi un programma che acquisisca due numeri decimali, variabili tra 0 a 255, e un operatore logico AND, OR oppure XOR, che li visualizzi in sistema binario e che mostri il risultato della loro combinazione, a seconda dell'operatore logico fornito, sempre in sistema binario.

Capitolo

I blocchi e le tabelle 12

12.1 Le ricerche nei blocchi

In aggiunta alle istruzioni che permettono lo spostamento di blocchi completi di memoria, si prenderanno ora in considerazione quelle che consentono la ricerca di un valore particolare, in un blocco di memoria. Prima di eseguire una qualsiasi delle istruzioni di ricerca in un blocco, l'accumulatore deve contenere il valore ricercato. La coppia di registri HL contiene l'indirizzo della prima locazione da verificare, mentre la coppia BC il numero delle locazioni di memoria tra le quali effettuare la ricerca.
La prima delle istruzioni è CPIR che significa "confronta, incrementa e ripeti". Essa ricerca attraverso i blocchi di memoria a partire dalla locazione indicata dalla coppia di registri HL. Dopo che il valore in ciascuna locazione è stato confrontato con l'accumulatore, il valore nella coppia di registri HL viene aumentato di 1, mentre quello nel registro BC è diminuito di 1. La ricerca continua fino a che non si ottiene il risultato cercato oppure il valore nel registro BC diventa zero; nel primo caso si azzera il flag di zero mentre non bisogna scordare che il registro BC diventando zero, non fa altrettanto.
Molto simile è la CPDR che significa "confronta, decrementa e ripeti" ed ha la stessa funzione della precedente, eccettuato il fatto che la coppia di registri HL deve puntare all'inizio l'indirizzo più alto del blocco. Ad ogni esecuzione diminuisce il valore nella coppia di registri HL. La figura 12.1 illustra un breve programma che ricerca in un blocco per trovare il valore 255 in una locazione di memoria. Esistono infine due istruzioni di ri-

```
10 REM go
20 REM org 23760
30 REM equ 32600 Blocco
35 REM Qui;defw 0
40 REM ld hl,Blocco
45 REM ld bc,100;!numero delle locazioni
50 REM ld a,255;!valore cercato
55 REM cpir;!cerca
57 REM !salta se non trova
60 REM jr nz,Fine
62 REM dec hl
65 REM ld (Qui),hl;!indirizzo del valore cercato
70 REM Fine;ret
75 REM finish
```

Figura 12.1

cerca di blocco che non si ripetono automaticamente; esse sono la CPI e la CPD e sono molto simili a quelle precedentemente illustrate, tranne che non si ripetono. Dopo ogni confronto il programmatore deve scrivere ulteriori istruzioni per controllare se è stato trovato in memoria un valore uguale a quello contenuto nell'accumulatore; in caso contrario sarà necessario effettuare un controllo per determinare se si è raggiunta la fine del blocco. Queste istruzioni vengono usate quando sono necessarie ulteriori elaborazioni.

12.2 I registri indice

Si è già verificato che la coppia di registri HL serve sovente da puntatore per permettere l'uso diretto dei dati dalla memoria. Esistono poi due altri registri a 16 bit impiegati come puntatori per i dati nella memoria, chiamati IX e IY. Essi sono anche conosciuti come registri indice e normalmente usati abbinati ai blocchi di memoria.
Sfortunatamente lo Spectrum utilizza il registro IY come puntatore al blocco di memoria che contiene le variabili di sistema, rendendolo così inaccessibile al programmatore in Assembler.
A livello pratico un registro indice serve all'indicazione della prima locazione in un blocco di memoria, mentre le successive sono individuate in base alla loro distanza da quella puntata. Lo spezzone di programma della figura 12.2 indica come viene impiegato un registro indice e il modo in

```
 10 REM go
 20 REM org 23760
100 REM equ 32600 Blocco
110 REM !
120 REM ld ix,Blocco
125 REM !
130 REM !
140 REM set 5,(ix+2);!terza locazione
145 REM !
150 REM !
160 REM dec (ix+4);!quinta locazione
165 REM !
170 REM !
180 REM ld a,(ix+8);!nona locazione
185 REM !
190 REM !
195 REM ret
200 REM finish
```

Figura 12.2

cui ci si riferisce a locazioni particolari precisando il registro indice e la distanza dalla locazione puntata. I registri indice vengono generalmente impiegati quando è necessario un riferimento ad un blocco di dati collegati, ad esempio a una tabella. Il registro IX può anche essere usato quando non ci si riferisce ad un blocco di dati indicando uno spostamento uguale a zero per puntare la locazione di memoria indicata dal registro indice.
Un registro indice può sostituire una coppia HL in qualsiasi istruzione che la impieghi come indice di una locazione di memoria, ma deve essere precisato uno spostamento.

12.3 Le tabelle di consultazione

Molto spesso dati collegati tra loro vengono memorizzati nel computer in forma di tabella o di elenco che può, in seguito, venire consultato per verificare se contiene un'informazione. Quando questa viene trovata in un elenco o in una tabella particolari, attiva un salto ad un'altra parte del programma; naturalmente se i dati si trovano in un'altra tabella, il salto indirizzerà verso un'altra parte del programma. Un programma che fa uso di tabelle di consultazione può essere formato da molte tabelle diverse fra loro, ciascuna con un salto collegato, o da una tabella singola con

```
 10 REM go
 15 REM org 40000
 20 REM !programma passo-passo
 25 REM !
 30 REM !spazio per le tabelle
 35 REM Duebyte;defb 6 14 16 22 24 30 32 38 40 46 48 54 56 62 198 2
03 206 211 214 219 222 230 238 246 254
 40 REM Trebyte;defb 1 17 33 34 42 49 50 58 194 195 196 202 204 205
 210 212 218 220 226 228 234 236 242 244 250 252
 45 REM Dueind;defb 9 25 35 41 43 57 225 227 229 233
 50 REM Quattroind;defb 33 42 34 203
 55 REM Quattro237;defb 34 42 67 75 83 91 115 123
 60 REM !spazio per eseguire un'istruzione
 65 REM Inistr;defb 0 0 0 0 201
 80 REM Contprog;defw 0
 85 REM !riserva registri per il programma in codice-macchina
 90 REM ld hl,0
 95 REM push hl;push hl;push hl;push hl
115 REM !immette l'indirizzo di partenza
120 REM call Immnum
122 REM ld hl,(Mem)
125 REM ld (Contprog),hl
130 REM ex de,hl;!il contatore e' DE
135 REM !trova il numero di byte per istruzione
140 REM Inizio;ld hl,Inistr
145 REM ld a,(de)
150 REM !verifica se il programma in codice-macchina e' finito
155 REM cp 201;!istruzione RET
160 REM jp z,Fine
165 REM !carica il primo byte
170 REM ld (hl),a
175 REM !e' un'istruzione con registri indice?
180 REM cp 221;!IX
185 REM jp z,Istrind
190 REM cp 253;!IY
195 REM jp z,Istrind
200 REM !istruzione col "237"
205 REM cp 237
210 REM jp z,Istr237
215 REM !istruzione a due byte?
220 REM ld hl,Duebyte
225 REM ld b,25;!lunghezza della tabella
230 REM Loopa;cp (hl)
235 REM jp z,Istrdue
240 REM inc hl
245 REM djnz Loopa
```

(continua)

```
250 REM !istruzione a tre byte
255 REM ld hl,Trebyte
260 REM ld b,26;!lunghezza della tabella
265 REM Loopb;cp (hl)
270 REM jp z,Istrtre
275 REM inc hl
280 REM djnz Loopb
285 REM !istruzioni a un byte
290 REM jp Esec
295 REM !istruzioni con registri indice
300 REM Istrind;inc de;!secondo byte
305 REM ld a,(de)
310 REM ld hl,Dueind
315 REM ld b,10;!lunghezza della tabella
320 REM Loopc;cp (hl)
325 REM jp nz,Succ
330 REM ld hl,Inistr+1
335 REM ld (hl),a
340 REM jp Esec
345 REM !continua la ricerca
350 REM Succ;inc hl
355 REM djnz Loopc
360 REM !istruzione a quattro byte?
365 REM ld hl,Quattroind
370 REM ld b,4;!lunghezza della tabella
375 REM Loopd;cp (hl)
380 REM jp z,Istrquattro
385 REM inc hl
390 REM djnz Loopd
395 REM !conserva l'istruzione a tre byte
400 REM ld hl,Inistr+1
405 REM ld (hl),a
410 REM inc de
415 REM inc hl
420 REM ld a,(de)
425 REM ld (hl),a
430 REM jp Esec
435 REM !conserva l'istruzione a quattro byte
440 REM Istrquattro;ld hl,Inistr+1
445 REM ld (hl),a
450 REM inc de
455 REM inc hl
460 REM ld a,(de)
465 REM ld (hl),a
470 REM inc de
475 REM inc hl
```

(continua)

```
480 REM ld a,(de)
485 REM ld (hl),a
490 REM jp Esec
495 REM !istruzione col "237"
500 REM Istr237;inc de
505 REM ld hl,Quattro237
510 REM ld b,8;!lunghezza della tabella
515 REM ld a,(de)
520 REM Loope;cp (hl)
525 REM jp z,Istrquattro
530 REM inc hl
535 REM djnz Loope
540 REM !e' un'istruzione a due byte
545 REM jp Istrduea
550 REM !istruzione a due byte
555 REM Istrdue;inc de
560 REM ld a,(de)
565 REM Istrduea;ld hl,Inistr+1
570 REM ld (hl),a
575 REM jp Esec
580 REM !istruzione a tre byte
585 REM Istrtre;inc de
590 REM ld hl,Inistr+1
595 REM ld a,(de)
600 REM ld (hl),a
605 REM inc de
610 REM inc hl;!terzo byte
615 REM ld a,(de)
620 REM ld (hl),a
630 REM !conserva l'indirizzo dell'istruzione successiva
640 REM Esec;inc de
650 REM ex de,hl
655 REM ld (Contprog),hl
660 REM pop hl;pop de;pop bc;pop af;!ripristina i registri
685 REM !esegue l'istruzione
690 REM call Inistr;push af;push bc;push de;push hl
695 REM !visualizza i registri
700 REM call Cancella
705 REM call Descr
707 REM ld a,22;rst 16;ld a,1;rst 16;ld a,0;rst 16
708 REM pop hl;pop de;pop bc;pop af
710 REM call Video;call Dueacapo;call Dueacapo
715 REM ld a,b
720 REM call Video
725 REM ld a,c
730 REM call Centro;call Dueacapo;call Dueacapo
```

(continua)

```
735 REM ld a,d
740 REM call Video
745 REM ld a,e
750 REM call Centro;call Dueacapo;call Dueacapo
755 REM ld a,h
760 REM call Video
765 REM ld a,l
770 REM call Centro;call Dueacapo;call Dueacapo
772 REM push hl
775 REM push af
780 REM pop hl
785 REM ld a,l
790 REM call Videofl;call Dueacapo
792 REM pop hl
795 REM push af;push bc;push de;push hl
820 REM ld hl,(Contprog)
825 REM ex de,hl
830 REM ld a,d
835 REM call Video
840 REM ld a,e
845 REM call Video
850 REM !azzera le locazioni per l'istruzione successiva
855 REM ld a,0
860 REM ld b,4
865 REM ld hl,Inistr
870 REM Loopf;ld (hl),a
875 REM inc hl
880 REM djnz Loopf
883 REM call Tastiera;ld hl,(Contprog);ex de,hl
885 REM jp Inizio
890 REM !subroutine
895 REM !
900 REM !cancella il video
905 REM Cancella;ld a,2;call 5633;call 3435
910 REM ret
940 REM !
945 REM !visualizza le descrizioni
950 REM Str;defs AccumulatoreBCDEHL S  Z  -  H  - P/V N  CPC
955 REM Descr;ld a,2
960 REM call 5633
995 REM !
1000 REM ld hl,Str
1005 REM ld b,12
1010 REM Descracc;ld a,(hl);!accumulatore
1015 REM rst 16
1020 REM inc hl
```

(continua)

```
1025 REM djnz Descracc
1030 REM call Dueacapo;call Dueacapo
1035 REM !
1040 REM ld b,3
1045 REM Descrreg;ld a,(hl)
1050 REM rst 16;ld a,6;rst 16
1055 REM inc hl
1060 REM ld a,(hl)
1065 REM rst 16
1070 REM call Dueacapo;call Dueacapo
1075 REM inc hl
1080 REM djnz Descrreg
1085 REM !descrizioni dei flag
1125 REM ld b,23
1130 REM Flags;ld a,(hl)
1135 REM rst 16
1140 REM inc hl
1145 REM djnz Flags
1185 REM call Dueacapo;call Dueacapo
1190 REM ld a,(hl)
1195 REM rst 16
1200 REM inc hl
1205 REM ld a,(hl)
1210 REM rst 16
1215 REM ret
1225 REM !
1230 REM Dueacapo;ld a,13;rst 16
1235 REM ld a,13;rst 16
1240 REM ret
1245 REM !
1250 REM Centro;push af
1255 REM ld a,6;rst 16
1260 REM pop af
1263 REM !visualizza il valore in esadecimale
1265 REM Video;push af;push bc;push af
1270 REM and 240
1275 REM ld b,4
1280 REM Shift;srl a
1285 REM djnz Shift
1290 REM call Cifra
1295 REM pop af
1300 REM and 15
1305 REM call Cifra
1310 REM pop bc;pop af
1315 REM ret
1320 REM !
```

(continua)

```
1325 REM !visualizza il valore dei flag
1330 REM Videofl;push af;push bc
1335 REM ld b,8
1340 REM Funo;sla a
1345 REM jp nc,Fzero
1350 REM push af
1352 REM ld a,32;rst 16
1355 REM ld a,49
1360 REM jr Fvid
1365 REM Fzero;push af
1367 REM ld a,32;rst 16
1370 REM ld a,48
1375 REM Fvid;rst 16;ld a,32;rst 16;pop af
1380 REM djnz Funo
1385 REM pop bc;pop af
1390 REM ret
1400 REM !aspetta che venga premuto un tasto
1405 REM Tastiera;ld bc,65278
1410 REM Filadopo;in a,(c);cpl;and 31;ret nz
1412 REM rl b;jr`c,Filadopo;jr Tastiera
1415 REM !visualizza una cifra esadecimale
1420 REM Cifra;cp 10;jp p,Lettera
1425 REM add a,48;jr Vis
1430 REM Lettera;add a,55
1435 REM Vis;rst 16;ret
1440 REM !immissione di un numero
1445 REM Immnum;ld hl,0
1446 REM ld (Mem),hl
1450 REM Loopnum;call Cinout
1455 REM cp 13;ret z
1460 REM call Perdieci;sub 48
1465 REM ld d,0;ld e,a;add hl,de
1467 REM ld (Mem),hl
1470 REM jr Loopnum
1471 REM Mem;defw 0
1475 REM !
1480 REM Perdieci;ld hl,(Mem);add hl,hl
1485 REM push hl;pop de;!copia in DE
1490 REM add hl,hl;add hl,hl
1495 REM add hl,de;ret
1500 REM !
1515 REM Cinout;call 703;call 4264
1520 REM cp 255;jr z,Cinout
1525 REM push af;rst 16
1530 REM Cinloop;call 703;call 4264
1535 REM cp 255;jr nz,Cinloop;pop af
```

(continua)

```
1540 REM ret
1544 REM Fine;rst 8;defb 13
1570 REM finish
2000 CLEAR 39999
2010 RANDOMIZE USR 58000
2020 STOP
2490 PAUSE 20
2500 RANDOMIZE USR 40080
```

Figura 12.3

un salto differente per ciascun elemento della tabella. Il programma della figura 12.3 è un esempio del primo tipo.
Il programma usa diverse tabelle e il ritrovamento di un dato in una tabella provoca un salto ad una diversa sezione del programma; ciascuna tabella possiede il suo particolare salto.
Oltre ad illustrare l'impiego delle tabelle di consultazione, questo programma costituisce un'utile routine per il programmatore in Assembler per far eseguire un programma in codice-macchina, un'istruzione per volta. Si inizia immettendo l'indirizzo di partenza del programma in codice-macchina, quindi viene eseguita un'istruzione e sono visualizzati i valori nei registri; si procede all'istruzione seguente quando viene premuto un tasto qualsiasi.

12.4 Le tabelle di salto

Una esigenza di programmazione molto comune è la verifica del valore di una variabile e il salto a una particolare sezione del programma, a seconda del valore della variabile.
Il metodo consiste nella memorizzazione della tabella di istruzioni di salto mediante i registri HL o IX come indici dell'inizio della tabella. Il valore della variabile serve per calcolare uno spostamento dal principio della tabella di salto; questo è aggiunto poi al valore del puntatore per fornire la posizione nella tabella. Le istruzioni:

JP (HL) o JP (IX)

sono impiegate per creare il salto alla parte desiderata del programma.
La figura 12.4 illustra l'uso di questa tecnica.

```
10 REM go
20 REM org 23760
30 REM !immettendo il numero del mese, ne visualizza il nome
40 REM call Immnum;ld hl,(Num);ld a,l
50 REM ld d,0;!azzera D
55 REM ld e,a;!pone il valore in E
60 REM sla a;!per 2
65 REM add a,e;!per 3
70 REM sub 3
75 REM ld e,a
80 REM ld hl,Tabsal
85 REM add hl,de
90 REM jp (hl);!salta alla tabella
95 REM !Tabella di salto
100 REM Tabsal;jp Gen
105 REM jp Feb
110 REM jp Mar
115 REM jp Apr
120 REM jp Mag
125 REM jp Giu
130 REM jp Lug
135 REM jp Ago
140 REM jp Set
145 REM jp Ott
150 REM jp Nov
155 REM jp Dic
160 REM !
165 REM Gen;ld hl,Ge
170 REM call Nome
175 REM ret
180 REM !
185 REM Feb;ld hl,Fe
190 REM call Nome
195 REM ret
200 REM !
205 REM Mar;ld hl,Mr
210 REM call Nome
215 REM ret
220 REM !
225 REM Apr;ld hl,Ap
230 REM call Nome
235 REM ret
240 REM !
245 REM Mag;ld hl,Mg
250 REM call Nome
```

(continua)

```
255 REM ret
260 REM !
265 REM Giu;ld hl,Gi
270 REM call Nome
275 REM ret
280 REM !
285 REM Lug;ld hl,Lu
290 REM call Nome
295 REM ret
300 REM !
305 REM Ago;ld hl,Ag
310 REM call Nome
315 REM ret
320 REM !
325 REM Set;ld hl,Se
330 REM call Nome
335 REM ret
340 REM !
345 REM Ott;ld hl,Ot
350 REM call Nome
355 REM ret
360 REM !
365 REM Nov;ld hl,No
370 REM call Nome
375 REM ret
380 REM !
385 REM Dic;ld hl,Di
390 REM call Nome
395 REM ret
400 REM !
405 REM Ge;defs Gennaio
410 REM defb 0
415 REM Fe;defs Febbraio
420 REM defb 0
425 REM Mr;defs Marzo
430 REM defb 0
435 REM Ap;defs Aprile
440 REM defb 0
445 REM Mg;defs Maggio
450 REM defb 0
455 REM Gi;defs Giugno
460 REM defb 0
465 REM Lu;defs Luglio
470 REM defb 0
475 REM Ag;defs Agosto
480 REM defb 0
```

(continua)

```
485 REM Se;defs Settembre
490 REM defb 0
495 REM Ot;defs Ottobre
500 REM defb 0
505 REM No;defs Novembre
510 REM defb 0
515 REM Di;defs Dicembre
520 REM defb 0
525 REM !
565 REM Nome;push hl;ld a,2;call 5633;pop hl
570 REM Rip;ld a,(hl)
575 REM cp 0;inc hl
580 REM jr z,Fine
585 REM rst 16
590 REM jr Rip
595 REM Fine;ld a,13
600 REM rst 16
605 REM ret
800 REM Immnum;ld hl,0;ld (Num),hl
810 REM Loopnum;call Cinout
820 REM cp 13;ret z
830 REM call Perdieci;sub 48
840 REM ld d,0;ld e,a;add hl,de
850 REM ld (Num),hl;jr Loopnum
860 REM Num;defw 0
900 REM Perdieci;ld hl,(Num);add hl,hl
910 REM push hl;pop de;add hl,hl
920 REM add hl,hl;add hl,de;ret
1000 REM Cinout;call 703;call 4264
1020 REM cp 255;jr z,Cinout
1040 REM push af;rst 16
1060 REM Cinloop;call 703;call 4264
1080 REM cp 255;jr nz,Cinloop;pop af
1100 REM ret
1110 REM finish
```

Figura 12.4

Il programma accetta un numero compreso tra uno e dodici, ed in base a questo visualizza il nome del mese; questo è naturalmente un impiego estremamente semplice di una tabella di salto poiché ogni sezione di programma ha la stessa lunghezza e sarebbe così più semplice calcolare l'inizio di ciascuna.

12.5 I numeri casuali

Per molti programmi di giochi è indispensabile generare numeri casuali al fine di introdurre nel gioco il fattore "fortuna" necessario; ciò può essere molto difficile con un programma in Assembler. La ROM dello Spectrum contiene un ottimo generatore di numeri casuali; tuttavia non è di semplice impiego e consigliabile solo a programmatori con una certa esperienza e una buona conoscenza del programma nella ROM. Esiste però una via semplice mediante la quale un qualunque programma in Assembler può usare i numeri casuali prodotti dalla ROM: ritornando al BASIC a metà programma, è possibile servirsi della funzione RND per generare un numero casuale, mettere con un POKE il valore in una locazione di memoria idonea ed infine ritornare al programma in Assembler tramite il comando USR.
Esiste un altro modo in cui un programma in Assembler riesce a produrre un numero che, anche se non propriamente casuale, può essere sfruttato come tale nella maggior parte dei casi. Uno dei registri nel microprocessore Z80, il registro R, serve al sistema per garantire che i dati non vengano persi dalla memoria; questo significa che il valore nel registro R cambia costantemente e, se caricato nell'accumulatore mediante l'istruzione:

```
LDA,R.
```

avremo a disposizione un valore compreso tra 0 e 255 ragionevolmente casuale. Se il vostro programma implica degli input dalla tastiera, che richiedono l'impiego di loop di lunghezza indeterminata durante l'attesa della pressione di un tasto, questo metodo probabilmente fornisce numeri casuali quanto il generatore di numeri casuali della ROM.
Benché, usando il registro R, venga fornito un numero compreso tra 0 e 255, non è necessario impiegare tutto questo intervallo. Si consideri un esempio di applicazione di tale metodo per rappresentare il lancio di un dado. Sono indispensabili dei numeri casuali da 1 a 6, perciò se si prendono i bit da 0 a 2 del numero nel registro R, si otterrà un numero variabile tra 0 e 7; scartando tutti gli zero e i sette, rimane un numero nell'intervallo richiesto. La figura 12.5 illustra un programma che simula il rotolare di due dadi finché non viene premuto un tasto.

```
 10 REM go
 20 REM org 23760
 25 REM !primo dado
 30 REM Inizio;ld a,r;!acquisisce il numero
 40 REM and 7;!00000111B
 45 REM cp 0;!scarta lo zero
 50 REM jp z,Inizio
 55 REM cp 7;!scarta il sette
 60 REM jp z,Inizio
 65 REM add a,48;!dal valore al codice
 70 REM ld c,a;!memorizzazione temporanea
 75 REM !secondo dado
 80 REM Sec;ld a,r
 85 REM and 7
 90 REM cp 0
 95 REM jp z,Sec
100 REM cp 7
105 REM jp z,Sec
110 REM add a,48
115 REM ld b,a
120 REM !visualizza i valori
125 REM ld a,2;push bc;call 5633;pop bc;!apre un canale
130 REM !PRINT AT 10,10
135 REM ld a,22;!AT
140 REM rst 16
145 REM ld a,10;rst 16
150 REM ld a,10;rst 16
155 REM ld a,b;rst 16
160 REM !PRINT AT 10,13
165 REM ld a,22;rst 16
170 REM ld a,10;rst 16
175 REM ld a,13;rst 16
180 REM ld a,c;rst 16
185 REM Tastiera;ld bc,65278
190 REM Filadopo;in a,(c);cpl;and 31
200 REM jr nz,Fine
212 REM rlc b;jr c,Filadopo
213 REM jp Inizio
215 REM Fine;ret
220 REM finish
```

Figura 12.5

12.6 Esercizio

Scrivete un programma per un semplice gioco del tipo "Master-Mind". Il computer dovrebbe produrre un numero casuale da 0 a 99 e il giocatore dovrebbe avere 5 possibilità per tentare di indovinare il numero. Dopo ogni tentativo, dovrebbe ricevere un messaggio che lo informi se il numero segreto è maggiore o minore di quello proposto.

Capitolo

Un po' più di aritmetica 13

13.1 I numeri a 16 bit

Tutti i numeri finora usati nei calcoli erano a 8 bit il che significa che si sono potuti svolgere calcoli aritmetici all'interno di un intervallo molto limitato.
Lo Spectrum ha però diversi registri a 16 bit il cui impiego offre indubbiamente un più ampio intervallo, sufficiente per gran parte dei problemi. Il microprocessore Z80 comprende alcune istruzioni aritmetiche a 16 bit che possono essere usate per operazioni con numeri a 32 bit, 48 bit o anche più lunghi.
Quando si eseguono calcoli a 16 bit la coppia di registri HL ha la funzione di accumulatore. La seguente istruzione di somma a 16 bit

```
ADD HL,ss
```

dove ss rappresenta uno dei registri a 16 bit BC, DE, HL o SP, aggiunge il valore nel registro ss a quello in HL e lascia in quest'ultimo il risultato. Oltre all'istruzione semplice ADD ne esiste una seconda per le addizioni a 16 bit, cioè l'istruzione ADC che ha la stessa struttura di ADD. La differenza tra le due consiste nel fatto che l'ADC, oltre ad aggiungere il valore nel registro ss a quello in HL, somma anche il valore nel flag di riporto; tutto questo significa che si può sommare un riporto di un'addizione precedente cosicché si riesce a lavorare a 32 bit o più.
La figura 13.1 illustra un programma che somma due numeri a 32 bit o 4 byte. Si noterà che ogni numero occupa quattro locazioni di memoria

```
10 REM go
20 REM org 23760
30 REM Primo;defw 500
40 REM defw 2000
50 REM Secondo;defw 150
60 REM defw 350
70 REM Risultato;defw 0
80 REM defw 0
90 REM !inizio del programma
100 REM ld hl,(Primo+2)
110 REM ld bc,(Secondo+2)
120 REM add hl,bc
130 REM ld (Risultato+2),hl
140 REM ld hl,(Primo)
150 REM ld bc,(Secondo)
160 REM adc hl,bc
170 REM ld (Risultato),hl
180 REM ret
190 REM finish
```

Figura 13.1

successive e bisogna prestare molta attenzione quando si trasformano i valori dei numeri a 32 bit in un unico numero decimale. Si ricordi che 32 bit possono contenere un numero da 0 a 4 294 967 295; per trasformare un numero racchiuso in quattro byte in un valore decimale semplice, il calcolo è il seguente:

valore totale =
= valore del primo byte * 16 777 216 + valore del secondo byte * 65536 + valore del terzo byte * 256 + valore del quarto byte

Mentre vi sono due istruzioni di addizione diverse per numeri a 16 bit c'è solo una istruzione di sottrazione, la SBC, la cui struttura è la seguente:

SBC HL,ss

dove ss è uno dei registri a 16 bit BC, DE, HL o SP. L'istruzione SBC ha l'effetto di sottrarre sia il valore del registro a 16 bit che quello del flag di riporto dal valore nel registro HL lasciando il risultato in quest'ultimo. L'istruzione SBC riesce a svolgere sottrazioni a 32 bit ma, prima di poter togliere dei numeri a 16 bit o i primi 16 bit di un numero a 32 bit, è necessario essere certi che il valore nel flag di riporto sia zero. Le due

istruzioni che consentono il cambiamento diretto del valore nel flag di riporto sono:

SCF che pone il valore uguale a 1, e
CCF che trasforma il valore nel flag di riporto nel suo opposto

La figura 13.2 illustra un programma per svolgere una sottrazione a 32 bit. Un punto degno di nota è che, nonostante le istruzioni ADC e SBC riguardino i flag di riporto, di superamento della capacità di memoria (overflow), di zero e di segno, contrariamente a qualunque aspettativa, l'istruzione ADD a 16 bit influenza solo il flag di riporto. Tutto questo significa che l'istruzione ADC è generalmente usata anche per l'addizione a 16 bit dopo essersi però assicurati che il flag di riporto contenga il valore zero.

```
 10 REM go
 20 REM org 23760
 30 REM Primo;defw 500
 40 REM defw 2000
 50 REM Secondo;defw 150
 60 REM defw 350
 70 REM Risultato;defw 0
 80 REM defw 0
 90 REM !inizio del programma
100 REM ld hl,(Primo+2)
110 REM ld bc,(Secondo+2)
114 REM scf
117 REM ccf
120 REM sbc hl,bc
130 REM ld (Risultato+2),hl
140 REM ld hl,(Primo)
150 REM ld bc,(Secondo)
160 REM sbc hl,bc
170 REM ld (Risultato),hl
180 REM ret
190 REM finish
```

Figura 13.2

13.2 I numeri a byte multipli

Si è appena parlato del modo in cui le istruzioni ADC e SBC permettono ai numeri multipli di 16 bit di essere sommati e sottratti. Esistono tuttavia delle versioni a 8 bit di queste istruzioni che consentono ai numeri

multipli di 8 bit di essere sommati e sottratti. La struttura delle istruzioni è:

ADC A,s
SBC A,s

dove s rappresenta un valore a 8 bit, un registro a 8 bit o anche il valore in una locazione di memoria puntato dalla coppia di registri HL. La figura 13.3 mostra un programma per eseguire un'addizione a byte multipli; il registro B contiene il numero dei byte.

```
 10 REM go
 20 REM org 23760
 30 REM !dati
 35 REM Lung;defb 3
 40 REM Primo;defb 12
 45 REM defb 30
 50 REM defb 100
 55 REM Secondo;defb 5
 60 REM defb 78
 65 REM defb 195
 70 REM Risultato;defb 0
 75 REM defb 0
 80 REM defb 0
 85 REM !azzera il flag di riporto
 90 REM scf
 95 REM ccf
100 REM !numero di byte in B
105 REM ld a,(Lung)
110 REM ld b,a
115 REM !punta al primo byte
120 REM ld de,Primo+2
125 REM ld hl,Secondo+2
130 REM ld ix,Risultato+2
135 REM !esegue la somma
140 REM Loop;ld a,(de)
145 REM adc a,(hl)
150 REM ld (ix+0),a
155 REM dec de
160 REM dec hl
165 REM dec ix
170 REM djnz Loop
175 REM ret
180 REM finish
```

Figura 13.3

13.3 BCD: sistema decimale in codice binario

Ogniqualvolta sono stati usati dei numeri, anche se inseriti dalla tastiera in forma decimale, è stata usata la rappresentazione binaria.
È molto difficile trasformare un numero binario in decimale e quindi in codici di carattere per l'output; fortunatamente esiste un'altra rappresentazione, conosciuta come BCD, ovvero sistema decimale in codice binario. In essa ciascuna cifra del numero decimale è rappresentata separatamente ed è espressa come un numero binario a 4 bit, chiamato "nibble". Ciascun registro a 8 bit o locazione di memoria può contenere due cifre di un numero in BCD.
La figura 13.4 indica il modo in cui può venire memorizzato un numero decimale a quattro cifre in due locazioni di memoria successive. Quando ci si serve di un nibble per rappresentare una cifra decimale, esso può contenere un numero nell'intervallo da 0 a 9; impiegando direttamente una rappresentazione binaria, quattro bit possono contenere numeri compresi tra 0 e 15, così la rappresentazione BCD spreca dello spazio.
Un altro inconveniente dei numeri in BCD ricorre durante lo svolgimento di operazioni aritmetiche. Il computer naturalmente si aspetta numeri binari puri e fornisce dei risultati errati con il BCD; questa questione sarà poi oggetto di discussione particolareggiata nel prossimo paragrafo. Per l'impiego dei numeri in BCD bisogna essere in grado di spostare le cifre dalla memoria all'accumulatore e viceversa: è possibile trasferire due cifre alla volta dei numeri in BCD oppure far uso delle istruzioni di shift o di rotazione per spostare un bit per volta, ma entrambi questi metodi

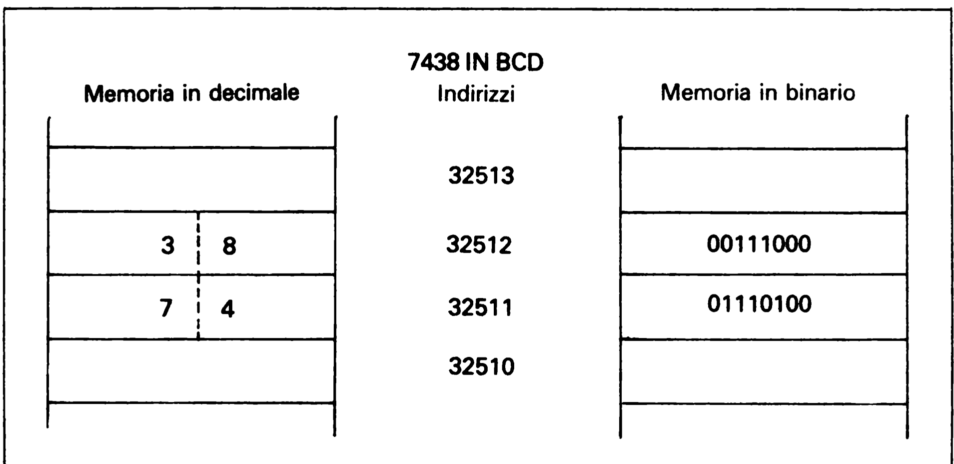


Figura 13.4

presentano degli inconvenienti.
Esistono due istruzioni in grado di fornire un movimento diretto tra la memoria e l'accumulatore tramite i nibble di dati; esse sono in realtà delle rotazioni che coinvolgono il nibble destro dell'accumulatore e i due nella locazione di memoria. Le due istruzioni in questione sono:

RLD	rotazione cifra a sinistra
RRD	rotazione cifra a destra

Prima che venga eseguita qualsiasi istruzione, il registro HL serve da indice per la locazione di memoria richiesta. La figura 13.5 illustra lo svolgersi delle istruzioni.

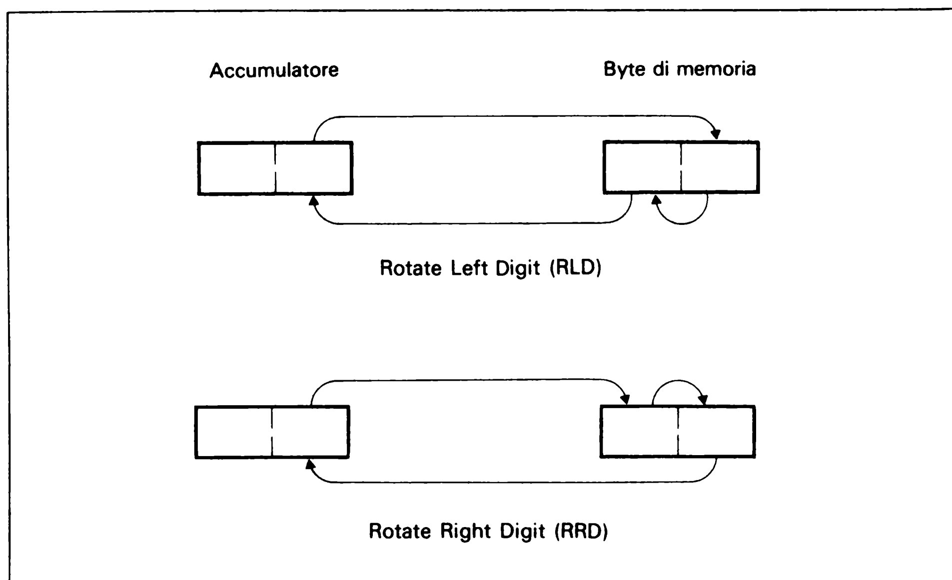


Figura 13.5

13.4 L'aritmetica del BCD

Se si considera la somma di numeri a due cifre in BCD mediante l'applicazione dell'aritmetica binaria e se ne interpretano poi i risultati in forma di numeri BCD, si riscontrerà che talvolta il risultato è esatto e altre è sbagliato:

```
34→00110100
51→01010001
   10000101→85  Risposta esatta
37→00110111
59→01011001
   10010000→90  Risposta sbagliata
36→00110110
55→01010101
   10001011→8?
```

In base all'ultima risposta, appare evidente che talvolta il risultato non può nemmeno essere rappresentato in cifra decimale perché i 4 bit rappresentano un numero più grande di 9.
Sono due i metodi applicati dai computer per garantire che l'aritmetica in BCD fornisca risposte esatte. Il primo offre un insieme completamente separato di istruzioni per l'aritmetica in BCD, mentre l'altro offre un modo di correzione dei risultati di calcoli effettuati in sistema binario sui numeri in BCD. Il processore centrale dello Spectrum impiega questo secondo metodo. Per i numeri in BCD le istruzioni aritmetiche (ADD, ADC, SUB o SBC) dovrebbero essere seguite dall'istruzione:

DAA

che corregge qualsiasi errore prodotto dall'uso dell'aritmetica binaria e fornisce la risposta nell'esatto BCD. Il programma nella figura 13.6 mostra l'input, l'addizione e l'output di numeri in BCD a due cifre. L'istruzione DAA serve anche per fornire una corretta aritmetica in BCD dopo le istruzioni INC, DEC, CP e NEG e opera esclusivamente sull'accumulatore.

```
10 REM go
20 REM org 23760
30 REM !somma di numeri a 4 cifre in BCD
35 REM Numero1;defw 0
40 REM Numero2;defw 0
45 REM Risult;defw 0
50 REM !acquisisce il primo numero
```

(continua)

```
55 REM ld hl,Numero1
60 REM call Duecifre
65 REM inc hl
70 REM call Duecifre
75 REM !acquisisce il secondo numero
80 REM inc hl
85 REM call Duecifre
90 REM inc hl
95 REM call Duecifre
100 REM !addizione
105 REM ld hl,Numero1
110 REM ld a,(hl)
120 REM ld hl,Numero2
130 REM add a,(hl)
135 REM daa;!correzione per BCD
140 REM ld hl,Risult
145 REM ld (hl),a
150 REM !seconda coppia di cifre
155 REM ld hl,Numero1+1
165 REM ld a,(hl)
170 REM ld hl,Numero2+1
175 REM adc a,(hl)
180 REM daa
185 REM ld hl,Risult+1
190 REM ld (hl),a
193 REM ret
195 REM !subroutine di acquisizione
200 REM Duecifre;push hl;call Cinout
205 REM sub 48;pop hl
210 REM ld (hl),a
215 REM push hl;call Cinout
220 REM sub 48;pop hl
225 REM rld
230 REM ret
1000 REM Cinout;call 703;call 4264
1020 REM cp 255;jr z,Cinout
1040 REM push af;rst 16
1060 REM Cinloop;call 703;call 4264
1080 REM cp 255;jr nz,Cinloop;pop af
1100 REM ret
1110 REM finish
```

Figura 13.6

13.5 Altre istruzioni

Si è parlato finora delle più utili istruzioni in Assembler; questo paragrafo ne presenterà di altrettanto vantaggiose. All'interno del processore centrale dello Spectrum c'è un'altra serie di registri a 8 bit, chiamati ausiliari; essi hanno gli stessi nomi dei registri principali e possiedono anche se non contemporaneamente il medesimo modo d'impiego. Le due serie si possono scambiare tramite l'istruzione:

```
EXX
```

Da quel momento tutte le istruzioni faranno riferimento alla seconda serie; eseguire di nuovo la stessa istruzione darà luogo ad uno scambio riportando i registri nella loro posizione originaria. Invece di scambiare tutta la serie di registri è possibile limitarsi a commutare l'accumulatore e il registro dei flag tramite l'istruzione:

```
EX AF,AF'
```

È già stato trattato il modo in cui il processore centrale invia dati all'esterno tramite l'istruzione OUT; in modo simile esso riceve le informazioni dall'esterno mediante l'istruzione IN. Entrambe hanno due strutture specifiche; la prima è:

```
IN A,(n)
OUT A,(n)
```

dove A rappresenta l'accumulatore, mentre n il numero a 16 bit della porta. La seconda struttura delle istruzioni è:

```
IN r,(C)
OUT r,(C)
```

dove r rappresenta uno dei registri a 8 bit, mentre il numero di porta si trova nella coppia BC.
Un'istruzione che può sembrare a prima vista di poca utilità per un programmatore è la NOP; essa significa "non operare" ed infatti ha l'effetto di non far compiere nessuna funzione al computer. I programmatori con una certa esperienza trovano che sia molto utile. Essa ha due principali impieghi; benché non faccia niente, richiede una certa quantità di tempo per operare e quindi è spesso usata per fare eseguire dei loop cronometrici accurati.
Dopo aver elaborato e assemblato un lungo programma, potrà presentar-

si l'eventualità di dover effettuare delle modifiche. Poiché il programma in codice-macchina è memorizzato in locazioni di memoria consecutive, diventa molto difficile inserire delle ulteriori istruzioni. Una tecnica vantaggiosa consiste nel separare le sezioni del programma tramite diverse istruzioni NOP; in tal modo sarà più facile inserire delle istruzioni o dei salti.
Infine l'ultima istruzione da considerare in questo paragrafo è quella di HALT. In un certo senso è molto simile a quella di STOP del BASIC, ma nello Spectrum il programma si ferma fino a che riceve un segnale di "interruzione", al termine della formazione di ogni immagine sul video.
In questo paragrafo si sono così completate le istruzioni in Assembler da studiare in modo particolareggiato. Abbiamo tralasciato solo un esiguo numero di istruzioni perché raramente utilizzate. Ora dovreste essere in grado di elaborare dei programmi in Assembler piuttosto soddisfacenti. Tutte le istruzioni utilizzabili nello Spectrum compaiono nell'Appendice A.

13.6 Esercizio

Scrivete un programma per inserire, aggiungere o sottrarre numeri BCD fino a sei cifre.
L'input sarà formato da un numero decimale, con o senza segno, seguito da un segno + o − e da un altro numero decimale; l'output dovrebbe apparire come nell'esempio seguente:

```
 123+456=579
772+-123=649
-123-456=-579
```

Capitolo

Ordinamento e selezione 14

14.1 L'ordinamento dei dati (sort)

Vorrei sfruttare questo capitolo finale per analizzare due metodi utilizzabili per ordinare i dati; ne esistono almeno quaranta differenti tra di loro, ma per gran parte dei programmi in Assembler, si può scegliere fra questi due. Il primo è semplice e di facile programmazione, relativamente lento, l'altro più avanzato, più difficile da programmare ma più veloce. Probabilmente è importante affermare che il miglior metodo per disporre i dati in modo ordinato nella memoria del computer è quello di collocarli nel corretto ordine al momento della loro introduzione nella macchina. Mediante veloci istruzioni di spostamento di blocchi e di ricerca negli stessi, ciascuna informazione può essere collocata al suo posto esatto, nel blocco dei dati, al momento dell'input.

14.2 Il sort a bolle

Il più semplice tipo di sort è quello a bolle. Il principio basilare di questo tipo di selezione è quello di confrontare le voci adiacenti nell'elenco. Se queste sono poste nell'ordine errato vengono scambiate ed in seguito paragonate alla coppia di voci successiva presente nell'elenco. Questo processo si ripete fino a che tutte le voci sono in ordine esatto. La figura 14.1 rappresenta il diagramma di flusso per un sort a bolle mentre la figura 14.2 un programma per realizzare un sort a bolle su una lista di da-

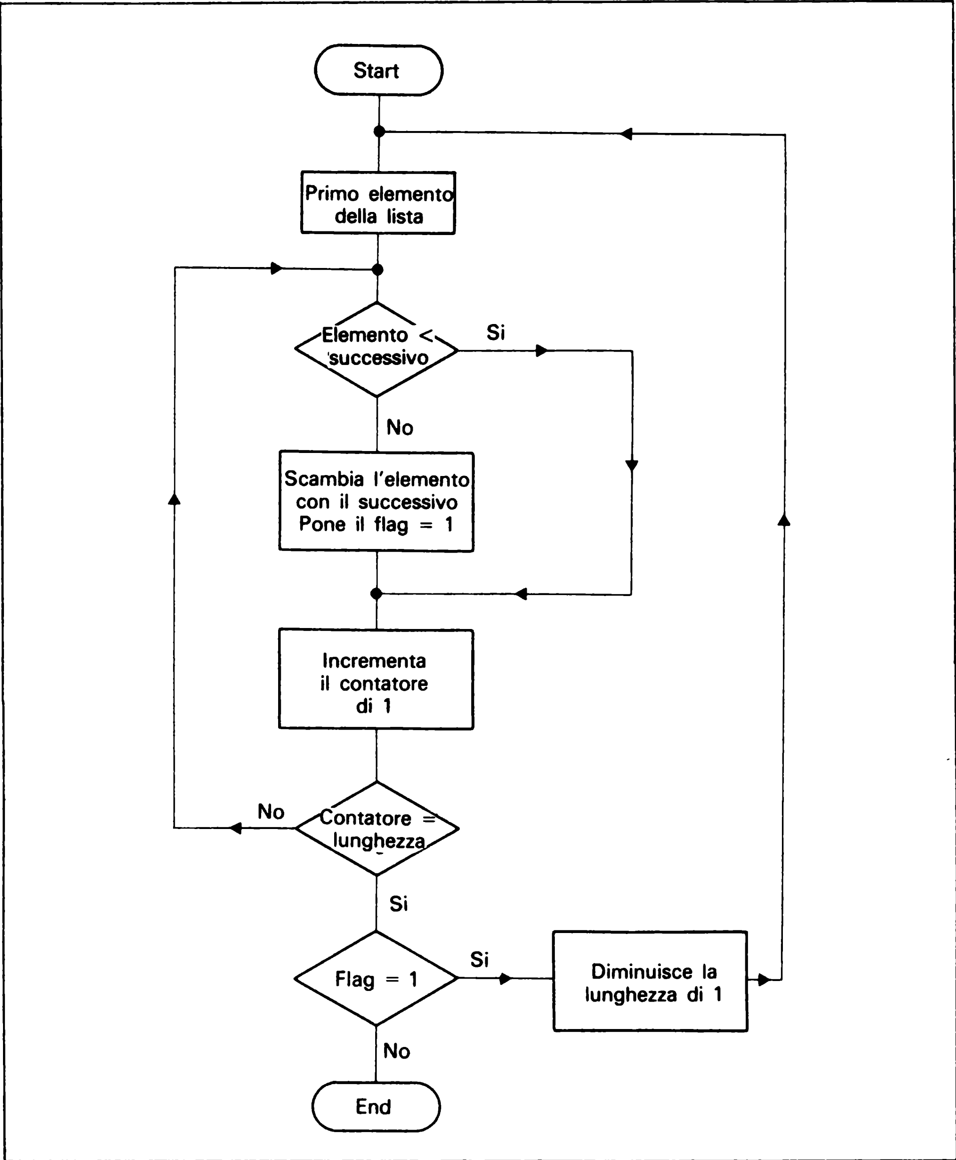


Figura 14.1

ti in un blocco di memoria. Il sort a bolle è di semplice programmazione e relativamente lento ma in Assembler si adatta a gran parte delle applicazioni.

```
10 REM go
20 REM org 23760
30 REM !sort a bolle
35 REM Temp;defb 0
40 REM ld a,(hl)
42 REM !B contiene la lunghezza della lista
44 REM !hl punta l'inizio della lista
46 REM Loopest;ld a,(hl)
50 REM inc hl;!elemento successivo
55 REM ld c,2;!contatore degli elementi
60 REM Loopint;cp (hl);!verifica l'ordine
65 REM jp m,Succ
70 REM !scambia gli elementi
75 REM ld d,a
85 REM dec hl
90 REM ld (hl),a
95 REM inc (hl)
100 REM ld (hl),d
105 REM ld e,1;!pone il flag uguale a uno
110 REM Succ;ld a,(hl)
115 REM inc hl
120 REM inc c
125 REM ld d,a
130 REM ld a,b
135 REM sub c;!verifica se la lista e' finita
140 REM jr nz,Loopint
145 REM ld a,e
150 REM cp 0;!se e' uguale a zero, la lista e' ordinata
155 REM ret z
160 REM dec b;!diminuisce la lunghezza di uno
165 REM jp Loopest
170 REM finish
```

Figura 14.2

14.3 Il sort di Shell

Il sort di Shell è sotto certi aspetti solo una forma più approfondita di quello a bolle, ma è molto più veloce e perciò migliore da usare quando si rende necessaria una maggiore velocità. Il sort di Shell, come quello a bolle, compie diversi passaggi al fine di riordinare i dati, ma si differen-

Elenco di 26 elementi			
Primo ciclo	Secondo ciclo	Terzo ciclo	Quarto ciclo
Differenza 13	Differenza 6	Differenza 3	Differenza 1
1 – 14	1 – 7	1 – 4	1 – 2
2 – 15	2 – 8	2 – 5	2 – 3
3 – 16	.	.	3 – 4
.	.	.	
.	.	.	
.	7 – 13	5 – 8	.
12 – 25	8 – 14	6 – 9	24 – 25
13 – 26	.	.	25 – 26
	.	.	
	.	.	
	19 – 25	22 – 25	
	20 – 26	23 – 26	

Figura 14.3

zia perché non confronta gli elementi adiacenti. Al primo passaggio la prima voce è paragonata a quella posta a metà dell'elenco, quindi la seconda viene sottoposta allo stesso trattamento con quella collocata immediatamente dopo la metà, e così via fino a raggiungere il termine dell'elenco. Al passaggio seguente la distanza fra le voci confrontate è dimezzata cosicché la prima è paragonata a quella situata ad un quarto della lunghezza dell'intero elenco. Al termine di ciascun passaggio la distanza fra le voci comparate è dimezzata fino al riordino della lista. La figura 14.3 illustra i confronti dei primi tre passaggi di un elenco composto da 26 voci. Le figure 14.4 e 14.5 rappresentano il diagramma ed il programma per il sort di Shell.

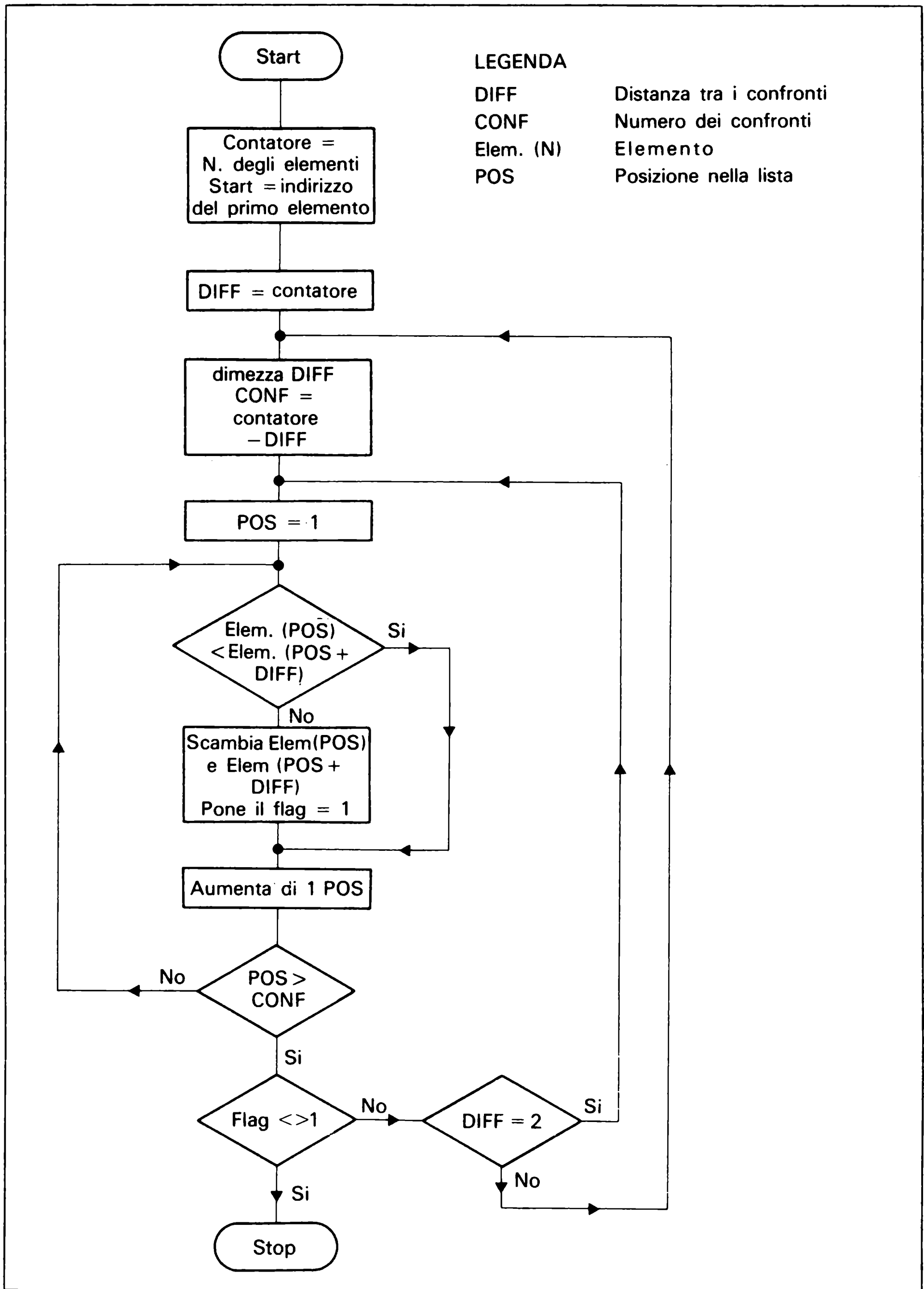

Start
LEGENDA
DIFF Distanza tra i confronti
CONF Numero dei confronti
Elem. (N) Elemento
POS Posizione nella lista
Contatore = N. degli elementi Start = indirizzo del primo elemento
DIFF = contatore
dimezza DIFF CONF = contatore − DIFF
POS = 1
Elem. (POS) < Elem. (POS + DIFF)
Si
No
Scambia Elem(POS) e Elem (POS + DIFF) Pone il flag = 1
Aumenta di 1 POS
No
POS > CONF
Si
Flag <>1
No
DIFF = 2
Si
No
Si
Stop

Figura 14.4

```
 10 REM go
 20 REM org 23760
 30 REM !sort di Shell
 32 REM !per liste fino a 255 elementi
 34 REM Cont;defb 0
 40 REM Flag;defb 0
 45 REM !HL punta il primo elemento
 50 REM ld d,0
 55 REM !B contiene il numero degli elementi
 60 REM ld a,b
 62 REM ld (Cont),a
 65 REM ld e,a
 70 REM !trova la differenza
 75 REM Loopa;sra e;!divide per due
 80 REM sub e
 85 REM ld b,a;!numero dei confronti
 87 REM xor a;!azzera A
 88 REM ld (Flag),a
 90 REM Loopb;ld a,(hl);!primo numero
 95 REM add hl,de
100 REM cp (hl);!secondo numero
105 REM jp m,Succ
110 REM !scambia gli elementi
115 REM ld c,(hl)
120 REM ld (hl),a
125 REM and a
130 REM sbc hl,de
135 REM ld (hl),c
140 REM ld a,1
145 REM ld (Flag),a
150 REM jr Succ1
155 REM Succ;and a
160 REM sbc hl,de
165 REM Succ1;inc hl
170 REM djnz Loopb
175 REM ld a,(Flag)
180 REM cp 0
185 REM ret z
190 REM ld a,e
195 REM cp 2
200 REM jp nz,Loopa
205 REM xor a
210 REM ld (Flag),a
215 REM ld a,(Cont)
```

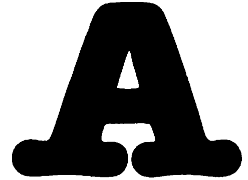
(continua)

```
220 REM dec a
225 REM ld b,a
230 REM ld (Cont)
235 REM jp Loopb
240 REM finish
```

Figura 14.5

Appendice

Sommario delle istruzioni Assembler



Questa appendice elenca tutte le istruzioni accettate dal microprocessore Z80 usato nello Spectrum. La tabella A.1 riassume gli effetti delle istruzioni sui bit nel registro dei flag. Sono indicate solo quelle istruzioni che riguardano i flag. Le tabelle rimanenti illustrano tutte le istruzioni ed il loro equivalente codice-macchina in sistema decimale.

Tabella A.1 Il registro dei flag

Istruzioni	Flag C	Z	P/V	S	N	H
ADD A	*	*	V	*	0	*
ADC A	*	*	V	*	0	*
SUB	*	*	V	*	1	*
SBC A	*	*	V	*	1	*
CP	*	*	V	*	1	*
NEG	*	*	V	*	1	*
AND	0	*	P	*	0	1
OR	0	*	P	*	0	1
XOR	0	*	P	*	0	0
INC m	–	*	V	*	0	*
DEC m	–	*	V	*	1	*
ADD HL	*	–	–	–	0	–
ADC HL	*	*	V	*	0	–
SBC HL	*	*	V	*	1	–
RLA, RLCA	*	–	–	–	0	0
RRA, RRCA	*	–	–	–	0	0
Rotazione e shift	*	*	P	*	0	0
RLD, RRD	–	*	P	*	0	0
DAA	*	*	P	*	–	*
CPL	–	–	–	–	1	1
SCF	1	–	–	–	0	0
CCF	*	–	–	–	0	–
IN	–	*	P	*	0	0
INI, IND, OUTI, OUTD	–	*	–	–	1	–
INIR, INDR, OTIR, OTDR	–	1	–	–	1	–
LDI, LDD	–	–	*	–	0	0
LDIR, LDDR	–	–	0	–	0	0
CPI, CPIR, CPD, CPDR	–	*	*	*	1	–
BIT	–	*	–	–	0	1
NEG	*	*	V	*	1	*

m solo operandi a 8 bit
* il flag è interessato
— il flag non è interessato
0 il flag è posto uguale a 0
1 il flag è posto uguale a 1
V il flag indica un overflow
P il flag indica parità

Tabella A.2 Istruzioni di LOAD a 8 bit–LD d, s.

	Provenienza dei dati										
Destinazione	A	B	C	D	E	H	L	(HL)	(IX+d)	(IY+d)	n
A	127	120	121	122	123	124	125	126	221 126 d	253 126 d	62 n
B	71	64	65	66	67	68	69	70	221 70 d	253 70 d	6 n
C	79	72	73	74	75	76	77	78	221 78 d	253 78 d	14 n
D	87	80	81	82	83	84	85	86	221 86 d	253 86 d	22 n
E	95	88	89	90	91	92	93	94	221 94 d	253 94 d	30 n
H	103	96	97	98	99	100	101	102	221 102 d	253 102 d	38 n
L	111	104	105	106	107	108	109	110	221 110 d	253 110 d	46 n
(HL)	119	112	113	114	115	116	117				54 n
(IX+d)	221 119 d	221 112 d	221 113 d	221 114 d	221 115 d	221 116 d	221 117 d				
(IY+d)	253 119 d	253 112 d	253 113 d	253 114 d	253 115 d	253 116 d	253 117 d				

	Provenienza dei dati					
Destinazione	(BC)	(DE)	(nn)	R	I	A
A	10	26	58 n n	237 95	237 87	
(BC)						2
(DE)						18
(nn)						50 n n
R						237 79
I						237 71

Tabella A.3 Istruzioni di LOAD a 16 bit

	Provenienza dei dati	
	nn	(nn)
BC	1, n, n	237,75, n, n
DE	17, n, n	237, 91, n, n
HL	33, n, n	42, n, n
SP	49, n, n	237, 123, n, n
IX	221, 33, n, n,	221, 42, n, n
IY	253, 33, n, n	253, 42, n, n

	Provenienza dei dati					
	BC	DE	HL	SP	IX	IY
(nn)	237 67 n n	237 83 n n	34 n n	237 115 n n	221 34 n n	253 34 n n

Tabella A.4 Istruzioni PUSH e POP

	AF	BC	DE	HL	IX	IY
PUSH	245	197	213	229	221 229	253 229
POP	241	193	209	225	221 225	253 225

Tabella A.5 Istruzioni di scambio

EXX	217
EX AF,AF'	8
EX DE,HL	235
EX (SP),HL	227
EX (SP),IX	221, 227
EX (SP),IY	253, 227

Tabella A.6 Istruzioni di blocco

LDI	237, 160
LDIR	237, 176
LDD	237, 168
LDDR	237, 184
CPI	237, 161
CPIR	237, 177
CPD	237, 169
CPDR	237, 185

Tabella A.7 Aritmetica generale

DAA	39
CPL	47
NEG	237, 68
CCF	63
SCF	55

Tabella A.8 Aritmetica a 8 bit e logica

	Provenienza dei dati										
	A	B	C	D	E	H	L	(HL)	(IX+d)	(IY+d)	n
ADD	135	128	129	130	131	132	133	134	221, 134, d	253, 134, d	198, n
ADC	143	136	137	138	139	140	141	142	221, 142, d	253, 142, d	206, n
SUB	151	144	145	146	147	148	149	150	221, 150, d	253, 150, d	214, n
SBC	159	152	153	154	155	156	157	158	221, 158, d	253, 158, d	222, n
AND	167	160	161	162	163	164	165	166	221, 166, d	253, 166, d	230, n
XOR	175	168	169	170	171	172	173	174	221, 174, d	253, 174, d	238, n
OR	183	176	177	178	179	180	181	182	221, 182, d	253, 182, d	246, n
CP	191	184	185	186	187	188	189	190	221, 190, d	253, 190, d	254, n
INC	60	4	12	20	28	36	44	52	221, 52, d	253, 52, d	
DEC	61	5	13	21	29	37	45	53	221, 53, d	253, 53, d	

Tabella A.9 Aritmetica a 16 bit

	Provenienza dei dati					
	BC	DE	HL	SP	IX	IY
ADD HL	9	25	41	57		
ADD IX	221 9	221 25		221 57	221 41	
ADD IY	253 9	253 25		253 57		253 41
ADC	237 74	237 90	237 106	237 122		
SBC	237 66	237 82	237 98	237 114		
INC	3	19	35	51	221 35	253 35
DEC	11	27	43	59	221 43	253 43

Tabella A.10 Istruzioni di rotazione e di shift

	Provenienza dei dati									
	A	B	C	D	E	H	L	(HL)	(IX+d)	(IY+d)
RLC(A)									221	253
		203	203	203	203	203	203	203	203	203
	7	0	1	2	3	4	5	6	d	d
									6	6
RRC(A)									221	253
		203	203	203	203	203	203	203	203	203
	15	8	9	10	11	12	13	14	d	d
									14	14
RL(A)									221	253
		203	203	203	203	203	203	203	203	203
	23	16	17	18	19	20	21	22	d	d
									22	22
RR(A)									221	253
		203	203	203	203	203	203	203	203	203
	31	24	25	26	27	28	29	30	d	d
									30	30
SLA									221	253
	203	203	203	203	203	203	203	203	203	203
	39	32	33	34	35	36	37	38	d	d
									38	38
SRA									221	253
	203	203	203	203	203	203	203	203	203	203
	47	40	41	42	43	44	45	46	d	d
									46	46
SRL									221	253
	203	203	203	203	203	203	203	203	203	203
	63	56	57	58	59	60	61	62	d	d
									62	62
RLD								237		
								111		
RRD								237		
								103		

Tabella A.11 Istruzioni di bit

L'istruzione BIT

Bit	0	1	2	3	4	5	6	7
	203	203	203	203	203	203	203	203
A	71	79	87	95	103	111	119	127
	203	203	203	203	203	203	203	203
B	64	72	80	88	96	104	112	120
	203	203	203	203	203	203	203	203
C	65	73	81	89	97	105	113	121
	203	203	203	203	203	203	203	203
D	66	74	82	90	98	106	114	122
	203	203	203	203	203	203	203	203
E	67	75	83	91	99	107	115	123
	203	203	203	203	203	203	203	203
H	68	76	84	92	100	108	116	124
	203	203	203	203	203	203	203	203
L	69	77	85	93	101	109	117	125
	203	203	203	203	203	203	203	203
(HL)	70	78	86	94	102	110	118	126
	221	221	221	221	221	221	221	221
(IX+d)	203	203	203	203	203	203	203	203
	d	d	d	d	d	d	d	d
	70	78	86	94	102	110	118	126
	253	253	253	253	253	253	253	253
(IY+d)	203	203	203	203	203	203	203	203
	d	d	d	d	d	d	d	d
	70	78	86	94	102	110	118	126

L'istruzione RES

Bit	0	1	2	3	4	5	6	7
	203	203	203	203	203	203	203	203
A	135	143	151	159	167	175	183	191
	203	203	203	203	203	203	203	203
B	128	136	144	152	160	168	176	184
	203	203	203	203	203	203	203	203
C	129	137	145	153	161	169	177	185
	203	203	203	203	203	203	203	203

L'istruzione RES

Bit	0	1	2	3	4	5	6	7
D	130	138	146	154	162	170	178	186
	203	203	203	203	203	203	203	203
E	131	139	147	155	163	171	179	187
	203	203	203	203	203	203	203	203
H	132	140	148	156	164	172	180	188
	203	203	203	203	203	203	203	203
L	133	141	149	157	165	173	181	189
	203	203	203	203	203	203	203	203
(HL)	134	142	150	158	166	174	182	190
	221	221	221	221	221	221	221	221
(IX+d)	203	203	203	203	203	203	203	203
	d	d	d	d	d	d	d	d
	134	142	150	158	166	174	182	190
	253	253	253	253	253	253	253	253
(IY+d)	203	203	203	203	203	203	203	203
	d	d	d	d	d	d	d	d
	134	142	150	158	166	174	182	190

L'istruzione SET

Bit	0	1	2	3	4	5	6	7
	203	203	203	203	203	203	203	203
A	199	207	215	223	231	239	247	255
	203	203	203	203	203	203	203	203
B	192	200	208	216	224	232	240	248
	203	203	203	203	203	203	203	203
C	193	201	209	217	225	233	241	249
	203	203	203	203	203	203	203	203
D	194	202	210	218	226	234	242	250
	203	203	203	203	203	203	203	203
E	195	203	211	219	227	235	243	251
	203	203	203	203	203	203	203	203
H	196	204	212	220	228	236	244	252

L'istruzione SET

Bit	0	1	2	3	4	5	6	7
	203	203	203	203	203	203	203	203
L	197	205	213	221	229	237	245	253
	203	203	203	203	203	203	203	203
(HL)	198	206	214	222	230	238	246	254
	221	221	221	221	221	221	221	221
(IX+d)	203	203	203	203	203	203	203	203
	d	d	d	d	d	d	d	d
	198	206	214	222	230	238	246	254
	253	253	253	253	253	253	253	253
(IY+d)	203	203	203	203	203	203	203	203
	d	d	d	d	d	d	d	d
	198	206	214	222	230	238	246	254

Tabella A.12 Istruzioni di salto, CALL e RETURN

	Condizione								
Istruzione	None	C	NC	Z	NZ	PE	PO	M	P
JP nn	195 n n	218 n n	210 n n	202 n n	194 n n	234 n n	226 n n	250 n n	242 n n
JR n	24 n	56 n	48 n	40 n	32 n				
JP (HL)	233								
JP (IX)	221 233								
JP (IY)	253 233								
CALL nn	205 n n	220 n n	212 n n	204 n n	196 n n	236 n n	228 n n	252 n n	244 n n
RET	201	216	208	200	192	232	224	248	240
DJNZ	16 n								
RETI	237 77								
RETN	237 69								

Tabella A.13 Istruzioni di Restart

RST 0	199
RST 8	207
RST 16	215
RST 24	223
RST 32	231
RST 40	239
RST 48	247
RST 56	255

Tabella A.14 Istruzioni di input/output

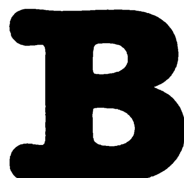
		Registro						
Istruzione	A,(n)	A,(C)	B,(C)	C,(C)	D,(C)	E,(C)	H,(C)	L,(C)
IN	219	237	237	237	237	237	237	237
	n	120	64	72	80	88	96	104
OUT	211	237	237	237	237	237	237	237
	n	121	65	73	81	89	97	105
INI	237, 162							
INIR	237, 178							
IND	237, 170							
INDR	237, 186							
OUTI	237, 163							
OTIR	237, 179							
OUTD	237, 171							
OTDR	237, 187							

Tabella A.15 Istruzioni varie

NOP	0
HALT	118
DI	243
EI	251
IMO	237, 70
IM1	237, 86
IM2	237, 94

Appendice

Lo ZX Spectrum Machine Code Assembler



B.1 L'impiego di un assemblatore

Tutti i programmi in questo libro sono stati ottenuti mediante il programma dello ZX Spectrum Machine Code Assembler. Questa appendice descrive l'uso di questo particolare programma assemblatore ma i concetti sono generalmente applicabili a qualsiasi altro programma assemblatore per lo Spectrum.
Esso è sostanzialmente un programma in codice-macchina caricato in testa alla memoria da un programma di supporto in BASIC. Una volta caricato il programma in codice-macchina, il programma in BASIC è automaticamente cancellato.
All'occorrenza il programma assemblatore entra in funzione grazie ad un comando del BASIC, cioè RANDOMIZE USR 26000 (per lo Spectrum 16K) o RANDOMIZE USR 58000 (per lo Spectrum 48K). Si è già notato che vengono usate due zone di memoria al momento della traduzione di programmi in Assembler: una è impiegata per memorizzare il programma in Assembler, mentre l'altra serve per quello in codice-macchina. Durante l'esecuzione del programma è più facile per il programma in codice-macchina riservare un'area di memoria in una istruzione REM all'inizio del programma BASIC. Quasi tutti i programmi illustrati in questo testo sono stati sviluppati con questo metodo. Il programma di supporto in BASIC deve partire con un'istruzione REM, contenente tanti caratteri quanti sono i byte di memoria impiegati dal programma in codice-macchina. Normalmente non si può prevedere il numero dei byte necessari prima di avere effettivamente tradotto il programma; si può espri-

mere una valutazione approssimata assegnando due byte a ciascuna istruzione in Assembler.

B.2 L'assemblatore ZX Spectrum

Lo ZX Spectrum Machine Code Assembler utilizza anche l'area di programma in BASIC per memorizzare le istruzioni in Assembler; tutte le istruzioni sono scritte nelle REM di un programma in BASIC. Dare uno sguardo ad alcuni dei programmi presenti in questo libro servirà a chiarire il principio.
In una REM può essere contenuta più di una istruzione, sempre che ciascuna di esse sia separata da un punto e virgola (;).
L'assemblatore accetta tutte le istruzioni dello Z80 illustrate nell'appendice A ed anche le direttive che saranno elencate nel prossimo paragrafo.
Le direttive sono istruzioni che non si trasformano in codice-macchina ma servono per fornire istruzioni al programma assemblatore; sovente sono chiamate pseudo-operazioni perché assomigliano alle istruzioni.
Le istruzioni presenti in questo testo sono state stampate in lettere maiuscole così da poterle facilmente riconoscere. Durante l'impiego dell'Assemblatore ZX Spectrum, è necessario introdurre tutte le istruzioni in lettere minuscole, come è indicato nel manuale dello Spectrum.
I numeri impiegati dallo ZX Spectrum possono essere sia in base decimale che esadecimale; questi ultimi sono preceduti da un segno di dollaro (ad esempio $ 1AB5). Le label sono usate come riferimento alle locazioni di memoria e l'assemblatore le trasformerà automaticamente nell'esatto indirizzo di memoria; le label possono avere una qualsiasi lunghezza, ma il primo carattere deve essere una lettera maiuscola. Le sole altre restrizioni circa le label, riguardano il fatto che non devono contenere i caratteri ")" oppure "+". Se usata all'inizio di un'istruzione come puntatore alla stessa, la label è considerata un'istruzione separata ed è seguita da un punto e virgola.
Esistono tre tipi di messaggi di errore che possono essere forniti durante la traduzione; se c'è un qualsiasi errore nelle direttive GO,FINISH o ORG, risulterà un messaggio di errore prima che parta l'assemblatore.
Un'istruzione errata viene indentificata con un messaggio lampeggiante che conterrà il numero di linea, il numero dell'istruzione nella linea ed il tipo di istruzione. Questo dovrebbe permettervi di trovare facilmente l'istruzione che contiene l'errore; sfortunatamente non è sempre altrettanto facile trovare l'errore vero e proprio.
Infine potrà comparire uno dei messaggi di errore del sistema. I possibili messaggi sono:

B "integer out of range"—un numero al di fuori dell'intervallo consentito.
2 "variable not found"—riferimento ad una label inesistente.
Q "parameter error"—istruzioni con errori di battitura
6 "number too big"—salto relativo al di fuori dell'intervallo consentito.

B.3 Le direttive

go— tutti i programmi in Assembler dello ZX Spectrum devono partire con questa istruzione ed essa deve trovarsi da sola in una REM

finish— è l'ultima istruzione in ogni programma ed anche questa deve trovarsi da sola in una REM

org— questa istruzione informa l'assemblatore circa la locazione di memoria da usare per il posizionamento del programma in codice-macchina. Nei modelli standard dello Spectrum 16K o 48K, **org** 23760 caricherà il programma in codice-macchina nel primo byte libero di una REM, al principio di un programma. **org** dovrebbe sempre essere la seconda istruzione del programma in Assembler ma può anche essere usata a metà del programma per far saltare l'assemblatore ad una nuova locazione nella memoria.

defb— consente di assegnare ad uno o più byte di memoria, a partire dall'indirizzo corrente, dei valori definiti nell'intervallo tra 0 e 255.

defw— permette di assegnare ad una parola (due byte) di memoria un valore definito nell'intervallo tra 0 e 65535.

defs— consente ad una stringa di caratteri di essere collocata nella memoria.

equ— è impiegata per assegnare una label ad una locazione di memoria. Si rivela utile quando l'indirizzo della locazione di memoria è posto al di fuori del programma in codice-macchina. Può servire a collegare spezzoni di programma in linguaggio macchina.

Appendice

C

Tabelle di conversione esadecimale-binario

Questo libro si è servito essenzialmente di numeri decimali poiché sono solitamente più comprensibili; tuttavia esistono dei casi in cui l'uso dei numeri binari o degli esadecimali ha più significato, in particolare quando si considerano gruppi di bit in un registro o in una locazione di memoria, oppure quando si trasforma un indirizzo di memoria a 16 bit in due numeri a 8 bit. Le tabelle seguenti presentano la conversione di numeri decimali ed esadecimali. La tabella C.1 fornisce la trasformazione per i numeri esadecimali fino a FF o 255 in decimali, mentre la tabella C.2, da usare con la precedente, comprende numeri fino a FFFF o 65535 in decimali.

Tabella C.1 Conversione di numeri esadecimali fino a FF o 255 in decimali

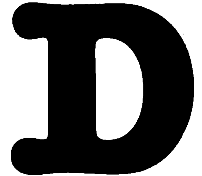
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
10	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
20	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
30	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
40	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
50	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
60	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
70	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
80	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
90	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
A0	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
B0	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
C0	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
D0	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
E0	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
F0	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255

Tabella C.2 Conversione di numeri esadecimali fino a FFFF o 65535 in decimali
(collegata con tab. C.1)

Esadecimale	Decimale	Esadecimale	Decimale
100	256	1000	4096
200	512	2000	8192
300	768	3000	12288
400	1024	4000	16384
500	1280	5000	20480
600	1536	6000	24576
700	1792	7000	28672
800	2048	8000	32768
900	2304	9000	36864
A00	2560	A000	40960
B00	2816	B000	45056
C00	3072	C000	49152
D00	3328	D000	53248
E00	3584	E000	57344
F00	3840	F000	61440

Appendice

L'assemblaggio manuale



D.1 Il metodo generale

Un programma elaborato in Assembler deve essere trasformato in codice-macchina prima di poter essere eseguito sul computer. Il modo più semplice per poter svolgere tale operazione consiste nell'impiego di un programma assemblatore. Per chiunque sia interessato all'elaborazione di un programma non piccolissimo, è consigliabile l'uso di un assemblatore. L'altro metodo di traduzione in codice-macchina consiste nell'usare le tabelle delle istruzioni e nel tradurre il programma a mano.
Per tradurre dal linguaggio assemblatore in codice-macchina, ogni istruzione deve essere cercata sulle tabelle delle istruzioni, come quelle nell'appendice A, per trovare la forma numerica dell'istruzione. Questo numero può essere binario, decimale o esadecimale; nel caso sia binario si è soliti trasformarlo in esadecimale o decimale.
Dopo la traduzione del programma in una lista di numeri, esso deve venire caricato nella memoria del computer. I numeri decimali possono venir caricati in memoria con l'uso di POKE mediante un programma molto semplice; i numeri esadecimali devono essere trasformati in decimali prima di poter usare il comando POKE per caricarli in memoria. Naturalmente prima di introdurre il programma nella memoria è necessario decidere dove deve essere caricato ed inserire delle istruzioni per tenere libera una zona di memoria per il programma.

D.2 Indirizzi e dati

Oltre a dover trasformare tutte le istruzioni in forma numerica, bisogna anche tradurre i dati in numeri della stessa base delle istruzioni. Il computer fa distinzione fra i dati e le istruzioni solo sapendo cosa dovrebbe trovarsi nella locazione di memoria successiva. Ad esempio, se l'indirizzo di partenza di un programma è la locazione 32000, il computer considererà il numero collocato in questa come un'istruzione; se il numero nella locazione 32000 è tradotto in un'istruzione che dovrebbe essere seguita dai dati, ad esempio la LD A,n, il computer prenderà il numero nella locazione 32001 come valore di n.
Uno dei problemi più importanti dell'assemblaggio manuale riguarda la traduzione dei numeri a 16 bit; essi devono essere inseriti nel computer come due numeri a 8 bit, e gli 8 bit più a destra devono essere caricati prima degli 8 bit più a sinistra. La figura D.1 illustra la trasformazione del numero decimale 32000 in due numeri a 8 bit, convertiti poi in decimali.
Applicando questo esempio, l'istruzione CALL 32000 andrebbe messa in tre locazioni di memoria consecutive come 205,0,125.

```
32000 = 0111110100000000B
      = 01111101/00000000
      = 125      0
32000 = 0, 125
```

Figura D.1

D.3 Le istruzioni di salto

Esistono due tipi di istruzioni di salto: i salti assoluti (istruzioni JP) e i salti relativi (istruzioni JR). Il metodo di trasformazione è lo stesso sia che i salti siano condizionati o incondizionati. I salti assoluti sono seguiti dall'indirizzo effettivo della locazione di memoria contenente l'istruzione che segue il salto. Se si sta effettuando un salto in avanti del programma, bisogna aspettare di averlo tradotto fino a quella istruzione, prima di riuscire a trovare l'indirizzo richiesto. Si ricordi che i byte dell'indirizzo sono rovesciati, come si è indicato nell'ultimo paragrafo. I salti relativi creano la gran parte dei problemi al momento dell'assemblaggio manuale

Assembler	Codice-macchina	
	126	
cp0	254	
	0	
jr z,Endit	40	salta avanti di 5 locazioni
	3	(5 − 2)
inc hl	35	
jr loop	24	salta indietro di 6 locazioni
	248	(− 6 − 2 in complemento binario)
Endit; ret	201	

Figura D.2

dei programmi; se il programma non opera in modo appropriato, probabilmente è meglio calcolare nuovamente i salti relativi. Il dato corretto per un'istruzione di salto relativo è il numero delle locazioni di memoria dall'istruzione di salto alla destinazione meno due. Se si tratta di un salto all'indietro del programma si avrà un numero negativo di locazioni, espresso con un numero a 8 bit in complemento binario. Viene sottratto due dallo spostamento perché, quando si esegue un'istruzione, il contatore del programma punta già la seguente. Il programma nella figura D.2 illustra la traduzione di due salti relativi.

D.4 Le istruzioni di bit

Poiché le istruzioni di verifica, variazione e ripristino dei bit singoli situati in un registro, dipendono da quest'ultimo e dal numero del bit, bisogna preoccuparsi dell'esatto impiego del codice dell'istruzione. Tutte le istruzioni di bit possiedono il codice 203 come primo byte.

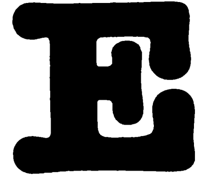
D.5 I registri indice

Le istruzioni dei registri indice sembrano più complesse di altre perché molte sono composte da 3 o 4 byte. Possono invece venire semplificate se si osserva che esse hanno le stesse istruzioni in codice-macchina di quelle

equivalenti per la coppia di registri HL, fatta eccezione per le istruzioni relative al registro IX che sono precedute dal byte 221 e quelle relative al registro IY precedute dal byte 253.
Entrambi i registri indice, quando sono usati come indici di memoria, devono comprendere un byte che indichi lo spostamento dalla locazione contenuta nel registro in questione, anche se tale spostamento è uguale a zero.

Appendice

I codici di carattere



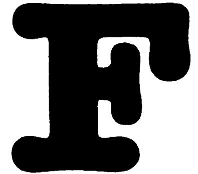
La tabella E.1 in questa appendice elenca i caratteri introducibili dalla tastiera e i loro relativi codici. A parte i caratteri grafici, essi sono quelli più usati anche per l'output.

Tabella E.1

32	(spazio)	56	8	80	P	104	h
33	!	57	9	81	Q	105	i
34	''	58	:	82	R	106	j
35	≠	59	;	83	S	107	k
36	$	60	<	84	T	108	l
37	%	61	=	85	U	109	m
38	&	62	>	86	V	110	n
39	'	63	?	87	W	111	o
40	(	64	@	88	X	112	p
41	)	65	A	89	Y	113	q
42	*	66	B	90	Z	114	r
43	+	67	C	91	[	115	s
44	,	68	D	92	/	116	t
45	–	69	E	93	]	117	u
46	.	70	F	94	↑	118	v
47	/	71	G	95	—	119	w
48	0	72	H	96	£	120	x
49	1	73	I	97	a	121	y
50	2	74	J	98	b	122	z
51	3	75	K	99	c	123	{
52	4	76	L	100	d	124	|
53	5	77	M	101	e	125	}
54	6	78	N	102	f	126	~
55	7	79	O	103	g	127	©

Appendice

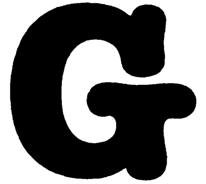
I caratteri di controllo della stampa



Codice	Tasto	Risultato
6		Stampa a metà schermo
8	←	Uno spazio indietro
13	ENTER	A capo
16	INK	Inchiostro (colore del carattere)
17	PAPER	Carta (colore dello sfondo)
18	FLASH	Carattere lampeggiante
19	BRIGHT	Carattere a doppia luminosità
20	INVERSE	Carattere inverso (negativo)
21	OVER	Sovrappone senza cancellare
22	AT	Controlla la posizione di stampa
23	TAB	Tabulazione

Appendice

Le subroutine della ROM



G.1 Il programma nella ROM

Nel momento stesso in cui si accende lo Spectrum, parte immediatamente un programma; esso è nella ROM ed occupa i primi 16k della memoria disponibile. L'intento di questo programma è quello di mettere il microprocessore in grado di poter comunicare con i vari dispositivi di input ed output usati dallo Spectrum ed inoltre di rendere possibile l'introduzione e l'esecuzione dei programmi in BASIC. Nello Spectrum in configurazione base gli input principali provengono dalla tastiera o dal registratore a cassette, mentre gli output sono inviati al televisore, all'altoparlante, al registratore e alla stampante. Il programma nella ROM è in codice-macchina ed è elaborato come una serie di subroutine; ciò significa che queste routine sono disponibili anche per un programma in Assembler. Questa appendice elencherà alcune delle routine più facili da usare e ne mostrerà il modo di impiego.

G.2 La stampa di un carattere

Il carattere il cui codice si trova nel registro A si visualizza sul canale corrente mediante la semplice istruzione:

```
RST 16
```

Per stampare nella parte superiore del video, per prima cosa deve venire aperto il canale esatto: la seguente sezione di programma stamperà un carattere nella parte superiore del video:

```
LD A,2
CAL 5633
LD A, Codice;! carattere da stampare
RST 16
```

Come indicato in precedenza, questa routine serve anche a modificare la posizione corrente di stampa e i colori temporanei mediante l'uso dei codici di controllo stampa forniti nell'appendice F.

G.3 La cancellazione del video

L'intero video può venire cancellato grazie al seguente spezzone di programma:

```
LD A,2
CALL 5633
CALL 3435
```

Esiste un'altra routine utilizzabile per cancellare solo una parte del video; questa elimina un dato numero di righe, contando a partire dal basso del video:

```
LD B,Righe;! Numero di righe da cancellare
CALL 3652
```

G.4 Gli spostamenti del video

Si può far muovere il video automaticamente caricando ripetutamente nella variabile di sistema SCR CT un numero maggiore di uno (probabilmente 255 rappresenta la scelta migliore). Si usino le istruzioni:

```
PUSH HL
LD HL,23692
LD (HL),255
POP HL
```

C'è una routine nella ROM che aiuta ad assegnare il numero di righe da spostare. Anche in questo caso, lo si otterrà contando a partire dal basso del video: infatti il valore caricato nel registro B è inferiore di uno rispetto alle righe da spostare:

```
LD B,Righe
CALL 3584
```

G.5 Il colore della cornice

Si può cambiare il colore della cornice inserendo il numero del colore richiesto nel registro A e quindi chiamando una subroutine ROM:

```
LD A,Colore
CALL 8859
```

G.6 I colori del video

Le variabili di colore sono memorizzate come dei byte nel file di attributo e nelle variabili di sistema ATTR-P, ATTR-T, MASK-P e MASK-T. Solitamente le routine della ROM dello Spectrum impiegano valori temporanei dei colori mentre altre, come quella di cancellazione video, usano i valori permanenti.
Gli attributi permanenti sono determinati mediante la modifica dei bit adatti della variabile di sistema ATTR-P; essa si trova alla locazione 23693 e gli attributi sono così memorizzati:

bit da 0 a 2	colore dell'inchiostro (del carattere)
bit da 3 a 5	colore della carta (dello sfondo)
bit 6	posto uguale a 1 indica BRIGHT (intensità maggiore)
bit 7	posto uguale a 1 indica FLASH (lampeggio)

Una volta designati i colori, come elementi permanenti o temporanei, possono essere usate le seguenti routine. Per copiare i valori permanenti nelle variabili di sistema temporanee, si usa l'istruzione:

```
CALL 3405
```

mentre per lo scopo inverso:

```
CALL 7341
```

G.7 L'input dalla tastiera

La routine di impiego più semplice è quella che controlla ripetutamente la tastiera fino a che si preme un tasto; se ciò avviene, viene inserito un valore nella coppia di registri DE.
Questa routine si usa tramite l'istruzione:

CALL 654

La routine principale per l'input del carattere è stata fornita nel capitolo 6.

G.8 Il suono

La routine che serve ad inviare una nota all'altoparlante può essere richiamata tramite l'istruzione:

CALL 949

Prima di usarla, al registro HL deve essere assegnato un valore che indica l'altezza della nota mentre al registro DE deve essere assegnato un valore che ne determina la lunghezza.

G.9 La stampante

Ci sono due routine nella ROM da usare con la stampante; la più semplice è la COPY che copia il contenuto del video alla stampante mediante l'istruzione:

CALL 756

I contenuti del buffer della stampante vengono stampati mediante l'istruzione:

CALL 3789

G.10 La grafica

Le routine di grafica per PLOT e DRAW possono essere usate per ottenere facilmente una grafica ad alta risoluzione. La routine PLOT esige che venga assegnato il valore della ascissa x del punto al registro C e quello dell'ordinata y al registro B prima di attivare la subroutine con l'istruzione:

CALL 8927

La DRAW si presenta più complessa poiché si possono usare i valori positivi e negativi di x e y. Prima di attivare la subroutine, i registri B e C dovrebbero contenere i valori assoluti rispettivamente di y e di x mentre D e E i segni rispettivamente di x e di y. Se x è positivo, D conterrà il valore 1 mentre se è negativo, conterrà −1; infine se x è zero, D manterrà il valore zero. L'istruzione per richiamare questa routine è:

CALL 9402

Indice analitico

I numeri si riferiscono ai paragrafi

Software

Cassetta disponibile in lingua inglese:

ZX Spectrum Machine Code Asembler

ISBN 07-084717-7

Questa cassetta è indispensabile per compilare i propri programmi scritti in Assembler, secondo le specifiche descritte nel libro di Tony Woods, *L'Assembler per lo ZX Spectrum.*

Nella stessa serie:

C.A. Street, *La gestione delle informazioni con lo ZX Spectrum*

J. Heilborn e R. Talbott, *Guida al Commodore 64*

R. Jeffries, G. Fisher e B. Sawyer, *Divertirsi giocando con il Commodore 64*

Di prossima pubblicazione:

G. Bishop, *Progetti hardware con lo ZX Spectrum*

H. Mullish e D. Kruger, *Il BASIC Applesoft*

N. Williams, *Inventa i tuoi giochi con lo ZX Spectrum*

H. Peckham, *Il BASIC pratico per l'IBM-PC*

H. Peckham, *Il BASIC pratico per il Commodore 64*

S. Nichols, *Tecniche avanzate in Assembler per giochi veloci con lo ZX Spectrum*

K. Skier, *L'Assembler per il VIC 20 e il Commodore 64*

S. Kamins e M. Waite, *Programmate meglio il vostro Apple*

L'Assembler è il linguaggio più vicino alla logica del computer e permette di realizzare programmi estremamente compatti e veloci.
Nel volume, che costituisce una completa e dettagliata introduzione alla programmazione in questo linguaggio, l'argomento è affrontato per gradi, così da permettere anche a chi è completamente digiuno in materia di arrivare ad una totale padronanza della sua sintassi.
Il testo è corredato di numerosi listati, immediatamente utilizzabili come programmi di utilità o come subroutine di programmi BASIC.
Complemento naturale a questo libro è lo **ZX Spectrum Machine Code Assembler**, un programma assemblatore - disponibile su cassetta - che permette di tradurre le istruzioni Assembler in codice macchina, utilizzabile direttamente dal microprocessore.
L'**Assembler per lo ZX Spectrum** è un libro indispensabile per chi voglia svincolarsi dai limiti imposti dal BASIC ed ottenere il meglio dal proprio computer.



Lire 18 000
(IVA 2% inclusa)

ISBN 88 7700 003 1

T. Woods

L'Assembler per lo ZX SPECTRUM

McGraw-Hill